www.galicea.org
fundacja@galicea.org
GALICEA
BUDUJEMY INNOWACYJNE SPOŁECZEŃSTWO

Uczymy się programować w Pythonie

Podręcznik programowania z użyciem języka Python

ISBN 978-83-63384-00-7

Galicea 2016

UCZYMY SIĘ PROGRAMOWAĆ W PYTHONIE

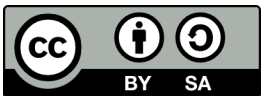
Otwarty podręcznik programowania

Jerzy Wawro

Ta książka jest do kupienia http://leanpub.com/pyprog

Wersja opublikowana 2017-07-03

ISBN 978-83-63384-00-7



I like to thank C.H.Swaroop for "A Byte of Python" - a free book on programming using the Python language. We used some of the materials from that book.

Spis treści

Wprowadzenia – o zawartości podręcznika

UWAGA! PODRĘCZNIK JEST KOMPLETNY,

ALE PRZED KOŃCOWĄ KOREKTĄ! Autor będzie wdzięczny za wszelkie uwagi.

Ponieważ niniejszy podręcznik ma służyć nauce programowania – język Python poznajemy niejako przy okazji. Osobom zainteresowanym systematycznym wykładem tego języka możemy polecić dostępne w języku polskim opracowania:

- *https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie*[1] – otwarty podręcznik Pythona
- *https://pl.python.org/docs/tut/tut.html*[2] - Przewodnik po języku Python autorstwa Guido van Rossum

Licencja

Podręcznik jest udostępniany na licencji *Creative Commons*[3] – zwanej w poniższym objaśnieniu warunków wykorzystania „Licencją". Podręcznik określa się jako „Dzieło". Użytkownik (czytelnik) korzystając z podręcznika automatycznie akceptuje warunki Licencji – stając się w ten sposób Licencjobiorcą.

Zgodnie z Licencją:

Licencjobiorca ma obowiązek zachować w stanie nienaruszonym wszelkie oznaczenia związane z prawno-autorską ochroną Dzieła oraz zapewnić, stosownie do możliwości używanego nośnika lub środka przekazu oznaczenie:

- imienia i nazwiska (lub pseudonimu, odpowiednio) Twórców, jeżeli zostały one dołączone do Utworu, oraz (lub) nazwę innych podmiotów jeżeli Twórca oraz (lub) Licencjodawca wskażą w oznaczeniach związanych z prawno-autorską ochroną Utworu, regulaminach lub w inny rozsądny sposób takie inne podmioty (np. sponsora, wydawcę, czasopismo) celem ich wymienienia ("Osoby Wskazane");

[1] https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie
[2] https://pl.python.org/docs/tut/tut.html
[3] https://creativecommons.org/licenses/by-sa/3.0/pl/legalcode

- tytułu Utworu, jeżeli został dołączony do Utworu;
- w rozsądnym zakresie URI, o ile istnieje, który Licencjodawca wskazał jako związany z Utworem, chyba że taki URI nie odnosi się do oznaczenia związanego z prawno-autorską ochroną Utworu lub do informacji o zasadach licencjonowania Utworu; oraz
- w przypadku Utworu Zależnego, oznaczenie wskazujące na wykorzystanie Utworu w Utworze Zależnym (np. "francuskie tłumaczenie Utworu Twórcy," lub "scenariusz na podstawie Utworu Twórcy").

Licencjobiorca może więc zgodnie z Licencją wykorzystywać i rozpowszechniać Dzieło pod warunkiem dołączenia kopii Licencji lub wskazania adresu strony internetowej (URI), pod którym znajduje się tekst Licencji do każdego egzemplarza Dzieła.

Licencjobiorca nie może oferować ani narzucać żadnych warunków lub ograniczeń nie uwzględnionych w Licencji. Nie może też udzielać sublicencji.

Niniejszy opis zawiera jedynie omówienie najważniejszych zapisów Licencji. Pełna jej treść jest dostępna na stronie: *Creative Commons Attribution-ShareAlike 4.0 International License*[4] Polskie tłumaczenie (do wersji 3.0): *https://creativecommons.org/licenses/by-sa/3.0/pl/legalcode*[5].

Prawa uzytkowania i powielania

Licencjobiorca ma prawo do wyświetlania tej książki, jej kopiowania i dystrybucji, tak długo, jak podaje oryginalnych autorów. Może modyfikować tekst książki, tworząc prace pochodne, o ile opisze różnice i opublikuje pracę na takiej samej licencji.

Zaproszenie do współpracy

Fundacja "**Galicea**", która jest inicjatorem stworzenia tego podręcznika, zachęca do współpracy przy jego rozwoju. Kontakt: fundacja@galicea.org[6]

Wykorzystane materiały i opracowania

Podstawą opracowania jest polskie tłumaczenie podręcznika „A Byte of Python" C.H. Swaroopa, udostępniony na licencji *Creative Commons*[7]. Niniejszy podręcznik udostępniamy na tej samej licencji. Jest więc on w świetle prawa „Utworem pochodnym" wobec wspomnianej książki (choć poza wykorzystanymi materiałami zawarto w nim wiele oryginalnych treści). Informacje o wykorzystanym utworze - zgodnie z warunkami licencji:

[4] http://creativecommons.org/licenses/by-sa/4.0/
[5] https://creativecommons.org/licenses/by-sa/3.0/pl/legalcode
[6] fundacja@galicea.org
[7] http://creativecommons.org/licenses/by-sa/4.0/

Autor: *Swaroop C H* http://www.swaroopch.com/about/[8]

Tytuł: „A Byte of Python”

Strona internetowa (URI):

- Podręcznik na gitbook: > *http://python.swaroopch.com*[9],
- Polskie tłumaczenie: > *http://python.edu.pl/byteofpython/*[10]

Oryginalny podręcznik może być pobrany z powyższych lokalizacji.

Zgodnie z przyjętą licencją, Licencjodawca może zastrzec, by wszystkie zmiany zostały zaznaczone i opisane. Tego zastrzeżenia nie C H Swaroop jednak nie uczynił. Biorąc pod uwag fakt, że niniejszy podręcznik nie jest wiernym tłumaczeniem oryginału, który posłużył jedynie za inspirację – zrezygnowano ze wspomnianych oznaczeń.

Przy opracowaniu niniejszego podręcznika celem nadrzędnym było uzyskanie jasnego i spójnego opisu sztuki programowania, niż wierność przekładu. Dlatego też zmieniono w stosunku do oryginału układ treści – przesuwając opisy mechanizmów trudniejszych i mniej uniwersalnych na sam koniec.

Przy opracowaniu rozdziału 7 (Deginiowanie i używanie funkcji) oraz trzech następnych wykorzystano wcześniejsze tłumaczenie podręcznika „Byte of Python” dostępne na stronie:

http://python.edu.pl/byteofpython/2.x/08.html[11]. Nie jest to jednak proste powielenie treści, ale opracowanie z uwzględnieniem zawartości pozostałych rozdziałów podręcznika.

Kod programów

Licencjonowanie kodu

Cały kod / skrypty znajdujące się w tej książce jest udostępniany na licencji BSD, chyba że zaznaczono inaczej. Kod ten jst dostępny na stronie: *http://python.owartaedukacja.pl*[12]

Równolegle opracowywany jest internetowy portal PyIDE, który jest dostepny na licencji GNU. Portal ten obsługuje wspomnianą wyżej stronę *http://python.otwartaedukacja.pl/*[13]

[8] http://www.swaroopch.com/about/
[9] http://python.swaroopch.com/
[10] http://python.edu.pl/byteofpython/
[11] http://python.edu.pl/byteofpython/2.x/08.html
[12] http://python.owartaedukacja.pl
[13] http://python.otwartaedukacja.pl/

Język angielski

Podręcznik jest tworzony w języku polskim. Jednak zdecydowano się stosować identyfikatory zmiennych bazujące na języku angielskim. Te kilkadziesiąt słów „współczesnej łaciny" łatwo opanować nawet komuś, kto nie zna języka angielskiego. Zastosowanie takiego rozwiązania ułatwi natomiast integracje z międzynarodowym środowiskiem programistów Pythona.

Z angielskiego pochodzą także tak zwane 'słowa kluczowe' - używane do zaznaczenia elementów struktury programu. Nie należy używać tych słów jako identyfikatorów. Pomimo, że Python rozróżnia duże i mal litery i teoretycznie identyfikator **For** nie będzie się mylił ze słowem kluczowym **for** – mogłoby to prowadzić do dodatkowych błędów w razie pomyłki.

Poniższa tabelka zawiera spis słów kluczowych:

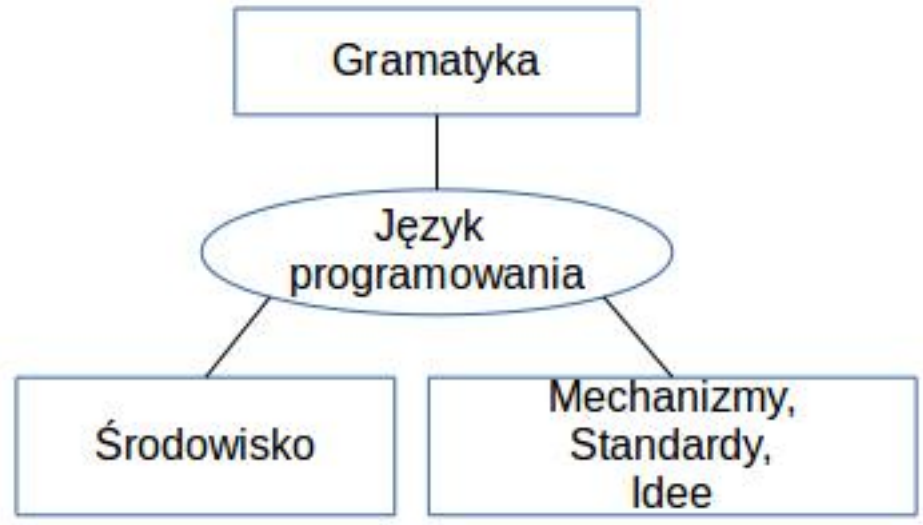
False	class	finally	is	raise
None	continue	for	lambda	return
True	def	from	not	try
and	del	global	or	while
as	elif	if	pass	with
assert	else	import		yield
break	except	in		

Myśleć jak programista Pythona

Ucząc się języka obcego musimy opanować podstawy gramatyki i słownictwo. Jednak opanowanie tego języka wymaga czegoś więcej. Jedni to nazywają duchem języka, inni myśleniem lub czuciem w obcym języku. Generalnie chodzi o opanowanie trudnego do opisania stylu wyrażania myśli.

Podobnie jest z językami programowania. Rozwój informatyki sprawił, że styl myślenia w języku programowania wyznaczają trzy elementy:

- Abstrakcyjne środowisko (świat) którego funkcjonowanie jest opisywane (definiowane) przez programy.
- Gramatyka – czyli sposób formułowania wypowiedzi (programów).
- Mechanizmy i idee programowania dostępne w danym języku.



Język programowania

Środowisko

Środowisko Pythona jest obiektowe. Oznacza to, że każdy element danych może mieć własności i metody. Własność to dana zawarta w obiekcie. Na przykład własnością łańcucha znaków (string) jest

reprezentowany przez ten obiekt napis. Metoda to sposób zmiany własności lub ich odczytywania. Na przykład łańcuch znaków zawiera funkcję zwracającą napis zmieniony na duże litery (upper). Metody to inaczej funkcje obiektu. Funkcje - czyli oznaczone nazwą fragmenty programów - zapisujemy w postaci: nazwa_funkcji(argument1, argment2,…)

Czyli po nazwie funkcji podajemy w nawiasach okrągłych argumenty (parametry) rozdzielone przecinkami.

Funkcja **f1** będąca metodą obiektu **obiekt1** może zostać wywołana po zapisaniu: **obiekt1.f1()**

Na przykład napis (łańcuch znaków) ma metody operowania na łańcuchach. W większości języków programowania chcąc zamienić znaki ‘b’ na ‘B’ w łańcuchu S napisalibyśmy coś w rodzaju: replace(S,’b’,’B’);

Czyli byłoby to polecenie operowania na łańcuchu znaków S przez zewnętrzną funkcję replace. W Pythonie łańcuch znaków jest obiektem, w którym zaimplementowano metody operowania na nim.

Przykład 1 (konsola):

Po uruchomieniu konsoli python (poleceniem python) wpisujemy (po znakach zachęty >>):

```
'123'.replace('2','a')
```

Wyjście:

```
'1a3'
```

Objaśnienie:

Łańcuch znaków (napis) oznaczamy cudzysłowami. Na przykład ‘123’ w programie oznacza napis 123 (można też stosować podwójny cudzysłów “). Ten napis jest obiektem, który posiada między innymi metodę zmiany *replace* z dwoma argumentami: co i na co ma być zmieniane. Zapis replace(‘2’,’a’) mówi, że każdą cyfrę 2 należy zamienić na literę a.

Konsola Pythona wyświetla wynik każdego działania – nie musimy więc dodatkowo używać polecenia wyświetlenia wyniku. Jednak gdybyśmy zapisali powyższe polecenie w pliku tekstowym – na przykład **objects.py** i wykonali powstały skrypt poleceniem: **python objects.py** – nie otrzymalibyśmy na ekranie żadnego wyniku. W programach trzeba podawać jawnie instrukcje wyświetlania (drukowania). Służy do tego funkcja print. Musimy więc zapisać:

```
print('123'.replace('2',"a"))
```

Przykład 1.1 (objects):

```
1 print('123'.replace('2',"a"))
2 n=1
3 print(n.__class__)
4 r=1.0
5 print(r.__class__)
```

Wyjście:

```
1 1a3
2 <type 'int'>
3 <type 'float'>
```

Objaśnienie:

Uwaga! własność __class__ jest obiektem i nie w każdym środowisku Python potrafi ją wyświetlić. Nie zdziw się więc, jeśli na wynik otrzymasz tylko pierwszy wiersz.

Poza użyciem metody *replace* powyższy przykład pokazuje użycie własności obiektów. W tym wypadku chodzi o własność __class__, która określa typ przechowywanych danych. Przy okazji widać, że użycie kropki dziesiętnej zmienia tym z liczby całkowitej (int) na rzeczywistą (float).

Język Python jest językiem operowania na obiektach.

Ponieważ zaś w postaci obiektów może być przedstawione prawie wszystko – spotyka się często zastosowanie Pythona do implementacji wtyczek do różnych programów (na przykład graficznych Gimp i Inkscape). Nawet Microsoft w końcu się przełamał i po latach promowania wyłącznie własnych technologii sięgnął po Pythona.

Mechanizmy

Większość języków programowania powstało po to, by precyzyjnie formułować przepis postępowania (wykonywania obliczeń). Rozwój inżynierii oprogramowania sprawił, że wprowadzono do nich mechanizmy ułatwiające obsługę błędów, podział na moduły, wydzielanie wspólnych elementów (biblioteki) etc... Python jest czymś w rodzaju zbioru najlepszych praktyk w tym zakresie. Pomimo, że jest to język uniwersalny, dąży się w nim do opisania tego, co ma być zrobione, a nie w jaki sposób. W powyższym przykładzie zapisujemy, że mają być zamienione cyfry 2 na litery a. W komputerze musi to polecenie zostać zamienione na przepis postępowania w postaci instrukcji odczytywania i zapisu komórek pamięci. W programach używamy jednak tak ogólnych zapisów (definicji) jak to tylko możliwe.

***Przykład 1.2* (sort):**

```
1 a=[1,5,4,2,0]
2 a.sort()
3 print(a)
4 print(a.sort())
```

Wyjście:

```
[0, 1, 2, 4, 5]
None
```

Objaśnienie:

Do zmiennej a wpisujemy listę liczb. Lista jest obiektem, który ma swoją metodę sortowania. Prawie każdy kurs programowania zawiera opis algorytmu (algorytmów) sortowania – czyli opis **jak to zrobić**. Jak widać w Pythonie to nie jest potrzebne. Wystarczy napisać co ma być zrobione.

Ostatni wiersz w powyższym przykładzie pokazuje, że należy odróżnić wynik działania funkcji (w tym wypadku metodę *sort()*) i wynik zwracany przez funkcję. Wywołanie *a.sort()* spowoduje posortowanie a, ale zwracany wynik jest pusty (None). Jeśli więc chcemy, aby w zmiennej b znalazły się posortowane liczby z a, to musimy napisać **b=a;b.sort()**, a nie **b=a.sort()**.

Bogactwo języka sprawia, że jedna z anegdot opowiada jak opisać stworzenie świata w Pythonie:

Przykład:

Jak stworzyć świat?

```
from przestrzen import stwarzanie
wszechswiat=stwarzanie()
```

Objaśnienie:

Czy to zadziała? Oczywiście! Pod warunkiem, że ktoś nam dostarczył implementacji – czyli moduł „przestrzen" z funkcją „stwarzanie" ;-).

Gdzie szukać takich implementacji? Zacząć trzeba na stronie *http://pypi.python.org*[14]. Klikając w „Browse package" (przeglądanie pakietów) uzyskujemy dostęp do tysięcy różnych implementacji. Pakiety z tej strony instalujemy na własnym komputerze poleceniem **pip**.

Ćwiczenie

Kiedy nastąpił biblijny dzień stworzenia świata?

[14] http://pypi.python.org/

1. Konwersję dat można znaleźć na stronie: *https://pypi.python.org/pypi/convertdate/*[15]

2. Przygotowujemy własne środowisko – żeby nie zaśmiecać serwera, uzyskać odpowiednią wersję Pythona i możliwość instalacji z internetu bez uprawnień administratora:

```
virtualenv -p /usr/bin/python warsztaty
source warsztaty/bin/activate
```

3. Instalujemy convertdate:

```
pip install convertdate
```

4. Uruchamiamy konsolę Pythona:

```
python
```

5. Testujemy:

```
convertdate.hebrew.from_gregorian(1, 1, 1)
```

Wyjście:

```
(3761, 10, 18)
```

Objaśnienie:

Zgodnie z kalendarzem hebrajskim, na dzień 1 stycznia pierwszego roku naszej ery przypadał 18 października 3751 roku (zob. strona *https://pl.wikipedia.org/wiki/Kalendarz_żydowski*[16]).

Gramatyka

Aby ułatwić komputerowi analizowanie tego co napisaliśmy, wprowadza się wyróżnione słowa – zwane słowami kluczowymi i inne znaki którymi zaznaczamy wyróżnione fragmenty tekstu. Python wyróżnia się tym, że zamiast stosowania zwyczajowych nawiasów {} odróżnia bloki tekstu tak zwanymi wcięciami, czyli spacjami o lewej stronie.

[15] https://pypi.python.org/pypi/convertdate/
[16] https://pl.wikipedia.org/wiki/Kalendarz_żydowski

Przykład 1.3:

```
1  # -*- coding: utf-8 -*-
2  # tabliczka mnożenia
3  cyfry=(0,1,2,3,4,5,6,7,8,9)
4  for a in cyfry:
5    for b in cyfry:
6      print('%s x %s = %s' % (a,b,a*b))
```

Objaśnienie:

Do zmiennej cyfry wstawiamy zbiór (tak zwaną krotkę – po angielsku ‘tuple’) liczb od 0 do 9.

Instrukcja for nakazuje zrobienie czegoś dla wszystkich elementów (w tym wypadku wszystkich cyfr). Słowo kluczowe **in** (z angielskiego ‘w’) mówi właśnie skąd mamy brać elementy.

Oznaczenie %s w napisie wskazuje miejsce, w które należy wstawić wartość. Po znaku % podaje się wstawiane wartości.

Na przykład

```
'%s x %s = %s' % (6,5,30)
```

spowoduje wyświetlenie

```
6 x 5 = 30
```

W wyniku powyższych instrukcji otrzymamy całą tabliczkę mnożenia.

Przy okazji poznajemy kolejny element Pythona: komentarze. To nie są fragmenty wykonywanego programu, ale tekst wpisany przez programistę – aby ułatwić czytanie programu.

Krótkie komentarze zaznacza się znakiem # na początku wiersza.

Ważna uwaga! Pomimo, że komentarze nie są wykonywane – są analizowane. Nie można w nich wpisywać czegokolwiek. W szczególności użyte znaki muszą być z dopuszczalnego alfabetu. Jakiego? To definiuje pierwszy wiersz programu:

```
# -*- coding: utf-8 -*-
```

Po słowie coding (kodowanie) podajemy standard znaków jaki chcemy użyć. Chcąc używać języka polskiego (literka ‘ż’ w komentarzu) – wybieramy standard utf-8.

[v1]

Python jako kalkulator

Nowoczesne kalkulatory pozwalają na wpisanie ciągu znaków zawierających wyrażenie i wykonanie wszystkich działań za jednym razem. Tak samo może działać konsola Pythona. Więcej na ten temat: *https://pl.python.org/docs/tut/node5.html*[17].

Podstawową różnicą jest to, że w wyrażeniach Pythona możemy używać zmiennych. Większość instrukcji programu zawiera *wyrażenia.* Prostym przykładem wyrazem jest `2 + 3` . Wyrażenie może być zbudowane z operatorów i argumentów.

Operatory to inaczej funkcje, które mogą być zapisane symbolami. Zamiast pisać mnożenie(1,2) zapisujemy `1*2`. Kolejność działań może być narzucona przez nawiasy. Na przykład: `(3+4)*3`. Dane na których są wykonywane działania (argumenty funkcji) nazywamy *argumentami* lub *operandami* (*https://pl.wikipedia.org/wiki/Operand*[18]).

Operatory

W Pythonie mamy dostępnych wiele różnych operatorów. Poniżej podano ich wykaz. Nie musisz wszystkich znać, by pisać programy. Pamiętaj, że zawsze możesz przetestować je używając interpretera Pythona (uruchamianego z konsoli poleceniem *python*). Na przykład, aby przetestować wyrażenie `2 + 3`, należy w wierszu poleceń interpretera Pythona wpisać:

```
>>> 2 + 3
```

Oto przegląd dostępnych operatorów:

- `+` (Plus)
  - Dodaje dwa obiekty
  - `3 + 5` daje `8` . `'a' + 'b'` daje `'ab'` .
- `-` (Minus)
  - Odejmuje od pierwszego argumentu drugi argument (po > prawej stronie). Jeśli pierwszy argument jest nieobecny, > zakłada się, że jest zerem.
  - `-5.2` Daje liczbę ujemną, a `50 - 24` daje `26` .
- `*` (Mnożenie)
  - Mnożenia dwóch liczb lub wielokrotne złączenie (powtórzenie) > łańcucha znaków.
  - `2 * 3` daje `6` . `'la' * 3` daje `'lalala'` .
- `**` (Potęga)

[17] https://pl.python.org/docs/tut/node5.html
[18] https://pl.wikipedia.org/wiki/Operand

  - Zwraca x do potęgi y
  - 3 ** 4 daje 81 (to jest 3 * 3 * 3 * 3)
- / (Dzielenie)
  - Podzielić x przez y
  - 13 / 3 daje 4.333333333333333
- // (Dziel i zaokrąglenie w dół)
  - Dzieli x przez y i zaokrągla wynik *w dół* do najbliższej liczby > całkowitej
  - 13 // 3 daje 4
  - -13 // 3 daje -5
- % (Modulo)
  - Zwraca resztę z dzielenia
  - 13 % 3 daje 1 . -25.5 % 2.25 daje 1.5 .
- << (Przesunięcie bitowe w lewo)
  - Przesuwa bity liczby w lewo o ilość pozycji określoną > drugim argumentem. (Każda liczba jest reprezentowana w pamięci > przez bity lub cyfry binarne, czyli 0 i 1)
  - 2 << 2 daje 8 . 2 jest w postaci binarnej jako 10.
  - Przesunięcie w lewo o 2 bity daje 1000, które reprezentuje > liczbę dziesiętną 8 .
- >> (Przesunięcie bitowe w prawo)
  - Przesuwa bity liczby w prawo o ilość pozycji określoną > drugim argumentem.
  - 11 >> 1 daje 5 .
  - 11 jest reprezentowane w postaci binarnej jako 1011. Po > przesunięciu o 1 bitw prawo zostaje liczba 101 która jest > zapisem wartości dziesiętnej 5 .
- & *Bitowa koniunkacja*[19]
  - Iloczyn liczona bit po bicie
  - 5 & 3 daje 1 (101 & 011 = 001).
- | (*Bitowa alternatywa*[20])
  - Alternatywa liczony bit po bicie:
  - 5 | 3 daje 7 (101 | 011 = 111)
- ^ (Bit-owy XOR)
  - Bitowe XOR liczb
  - 5 ^ 3 daje 6 (101 | 011 = 110)
- ~ (Bit-owa negacja)
  - Negacja logiczna x daje -(x + 1)
  - \~5 daje -6 . Więcej szczegółów na > *http://stackoverflow.com/a/11810203*[21]
- < (Mniejsze niż)
  - Sprawdza czy x jest mniejsze od y. Wszystkie operatory > porównania zwracają True (prawda) lub False (fałsz). Uwaga – > wielkość liter w zapisie True i False ma znaczenie!
  - 5 < 3 daje False i 3 < 5 daje True .
  - Porównania mogą być dowolnie łączone: 3 < 5 < 7 daje > True .

[19] https://pl.wikipedia.org/wiki/Operator_bitowy
[20] https://pl.wikipedia.org/wiki/Operator_bitowy
[21] http://stackoverflow.com/a/11810203

- > (Większe niż)
  - Sprawdza czy x jest większe od y
  - 5 > 3 powraca True . Jeśli oba argumenty są liczbami, są > najpierw konwertowane do wspólnego typu. W przeciwnym razie > zawsze zwraca False .
- <= (Mniejszy lub równy)
  - Sprawdza czy x jest mniejsza niż lub równa y
  - x = 3; y = 6; x <= y zwraca True
- >= (Większy lub równy)
  - Sprawdza czy x jest większe niż lub równe y
  - x = 4; y = 3; x >= 3 zwraca True
- == (Równy)
  - Sprawdza czy obiekty są równe
  - x = 2; y = 2; x == y zwraca True
  - x = 'str'; y = 'stR'; x == y zwraca False
  - x = 'str'; y = 'str'; x == y zwraca True
- != (Nie równe)
  - Sprawdza czy obiekty nie są równe
  - x = 2; y = 3; x != y zwraca True
- not (logiczne NIE)
  - Jeśli x jest True, to zwraca False . Jeśli x jest False, to > zwraca True .
  - x = True; not x zwraca False .
- and (logiczne I)
  - x and y zwraca False jeśli x jest False, inaczej zwraca wartość y
  - x = False; y = True; x and y zwraca False, ponieważ x > jest fałszywe. W tym przypadku, Python nie sprawdza y, > ponieważ wiadomo, że lewa strona wyrażenia "i" jest False co > oznacza, że całe wyrażenie jest False niezależnie od > innych wartości. To się nazywa skróconą ewaluacją wyrażeń i > jest ważne – bo na przykład jeśli nawet y nie jest poprawnie > zdefiniowane, to nie pojawi się błąd. Z tego względu poprawnym > jest **(1>2) and (1/0==0)**.
- or (logiczne LUB)
  - Jeśli x jest True, to zwraca True, w przeciwnym wypadku zwraca wartość y
  - x = True; y = False; x or y zwraca True . Skrócona ewaluacja > wyrażeń ma zastosowanie także tutaj.
    Łączenie przypisywania i operacji matematycznych

Gdy mamy wykonać działanie matematyczne na zmiennej a następnie przypisać wynik działania z powrotem do tej zmiennej, wykonujemy to następująco:

```
a = 2
a = a * 3
```

ten zapis można skrócić jako:

```
a = 2
a *= 3
```

Ogólnie zamiast

```
zmienna = zmienna operator wyrażenie
```

zapisujemy

```
zmienna operator= wyrażenie.
```

Kolejność wartościowania

Komputery działają sekwencyjnie. W szkole nas uczono, że w wyrażeniach takich jak `2 + 3 * 4`, najpierw robimy mnożenie. Mnożenie ma bowiem wyższy priorytet. W języku Python również ustalono priorytety. Zaleca się jednak, aby nie polegać na tym, tylko stosować nawiasy (chodzi o jednoznaczność i czytelność zapisu). Poniższa tabelka zawiera priorytety operatorów:

Operator	**Opis**
lambda	Wyrażenie lambda
or	Logiczne OR (lub)
and	Logiczne AND (i)
not *x*	Logiczne NOT (nie)
in, not in	Testy przynależności
is, is not	Testy tożsamości
<, <=, >, >=, <>, !=, ==	Porównania
|	Bitowe OR (lub)
^	Bitowe XOR (różnica symetryczna)
\&	Bitowe AND (i)
<<, >>	Przesunięcia
+,	Dodawanie i odejmowanie
*, /, %	Mnożenie, dzielenie, reszta z dzielenia
+*x*, *x*	Identyczność, negacja
\∼*x*	Bitowe NOT (nie)
**	Potęgowanie
x.atrybut	Odwołanie do atrybutu

Operator	Opis
x[*indeks*]	Odwołanie do indeksu
x[*indeks*:*indeks*]	Wykrojenie
f(*argumenty...*)	Wywołanie funkcji
(*wyrażenia...*)	Powiązanie lub drukowalna forma krotki
[*wyrażenia...*]	Drukowalna forma listy
{*klucz*:*dana...*}	Drukowalna forma słownika
‘*wyrażenia...*‘	Konwersja napisowa

Operatory o tym *samym priorytecie* są wymienione w tym samym wierszu w powyższej tabeli. Na przykład, + i -ma ten sam priorytet.

Zmiana kolejności działań

Aby uczynić wyrażenia bardziej czytelne, możemy użyć nawiasów. Na przykład, `2 + (3 * 4)` jest zdecydowanie łatwiejsze do zrozumienia niż `2 + 3 * 4`, co wymaga znajomości priorytetów operatorów. Jak ze wszystkim, nawiasy powinny być wykorzystywane racjonalnie – nie należy stosować ich w nadmiarze - jak w `(2 + (3 * 4))` .

Podstawową funkcją nawiasów jest jednak modyfikacja kolejności działań. Na przykład, jeśli chcesz wykonać dodawanie przed mnożeniem, to można napisać: `(2 + 3) * 4` .

*. Łączność

Operatory są zwykle stosowane z lewej do prawej. Oznacza to, że operatory o tym samym priorytecie są stosowane od lewej do prawej. Na przykład, `2 + 3 + 4` jest równoważny z `(2 + 3) + 4` .

*. Wyrażenia

Pokażemy użycie prostych wyrażeń do wyliczenia powierzchni prostokąta.

Przykład

```
length = 5
breadth = 2
area = length * breadth
print(u'Powierzchnia prostokąta = %s' % area)
print(u'Obwód = %s' % (2 * (length + breadth)) )
perimeter = 2 * (length + breadth)
print(u'Obwód = %s' % perimeter )`
```

Wyjście:

```
('Powierzchnia prostok\\xc4\\x85ta =', 10)
('Obw\\xc3\\xb3d =', 14)
```

Ten przykład pokazuje kolejny problem z formatowaniem wyjścia. Skąd Python ma wiedzieć, że napis jest po polsku? Samo wskazanie kodowania (zob. `# -*- coding: utf-8 -*-` z przykładu 1.3) nie wystarczy. Przed łańcuchem musimy podać literkę u (od unicode). Przy okazji robimy formatowanie jak w przykładzie 1.3:

Przykład 2.1: (expression)

```
1 length = 5
2 breadth = 2
3 area = length * breadth
4 print(u'Powierzchnia prostokąta = %s' % area)
5 print(u'Obwód = %s' % (2 * (length + breadth)) )
6 perimeter = 2 * (length + breadth)
7 print(u'Obwód = %s' % perimeter )
```

Wyjście:

```
Powierzchnia prostokąta = 10
Obwód = 14
Obwód = 14
```

Jak to działa

Długość i szerokość prostokąta są przechowywane w zmiennych odpowiednio length i breadth. Używamy ich do wyliczenia wyrażeń określających powierzchnię i obwód prostokąta. Wstawiamy do zmiennej area wynik wyrażenia length * breadth, aby następnie wydrukować go za pomocą funkcji print (druk). W tym przypadku obwodu możemy podać w funkcji drukowania bezpośrednio wyrażenie 2 * (length + breadth) – bez przechowywania wyniku w zmiennej. Jednak należy zwrócić uwagę na to, że operator % oczekuje dokładnie tylu elementów ile zaznaczono (%s) w formatowanym łańcuchu znaków. Dlatego albo należy ująć wyrażenie w nawiasy, albo zastosować zmienną (perimeter).

Podsumowanie

Poznaliśmy sposób konstruowania wyrażeń - podstawowych elementów w dowolnym programie. Dzięki temu konsola Pythona może być używana jako kalkulator.

Przetwarzanie niewielkich ilości danych

Większość działań jakie wykonują programy jest trywialnie prosta. Jednak zanim pojawiły się nowoczesne (obiektowe) programowania – nawet tak banalne zadania jak posortowanie danych, czy wybranie najmniejszego elementu wymagały pisania wielu linijek kodu i nastręczały wiele okazji do przykrych błędów. W tych czasach stosowano powszechnie metodologię programowania od ogółu do szczegółu. Weźmy prosty przykład: podzielić dzieci na kolonii na dwie równe grupy, biorąc pod uwagę wiek dzieci. Załóżmy że dysponujemy listą zawierającą imię i nazwisko oraz wiek każdego dziecka (zapisane wprost z Excela w formacie CSV).

Jak to zrobić?

1) wczytać_listę:

2) posortować_zbiór;

3) wybrać_element_środkowy;

Teraz każdy z tych elementów uzupełniamy o szczegóły. Na przykład przy wybieraniu środkowego elementu należy ustalić czy ilość elementów jest nieparzysta. Jeśli nie – musimy zdecydować, czy wziąć element o numerze n/2, czy też n/2+1 (gdzie n = ilość elementów). Dla wprawnego programisty jakieś pół godziny roboty.

Jak z tym zadaniem zmierzy się programista Pythona?

1. Poszuka w internecie hasła: Python+CSV. Na przykład: https://pythonprogramming.net/reading-csv-files-python-3/. Kopiuj i wklej + drobna modyfikacja i mamy program wczytujący listę zawierającą wiek dzieci.

2. Teraz pora na odszukanie elementu środkowego, czyli mediany. Na pewno trafimy na funkcję numpy.median(). I tu niespodzianka – nie musisz nic sortować!

A oto efekt 10 minut pracy (razem z szukaniem błędu wynikłego z przeoczenia konieczności konwersji danych):

Przykład 3.1: **mediana**

```
1 import csv
2 import numpy
3 ages=[]
4 with open('~/lista.csv') as csvfile:
5   readCSV = csv.reader(csvfile, delimiter=;)
6   for row in readCSV:
7     ages.append(int(row[2]))
8 m=numpy.median(ages)
9 print(m)
```

Objaśnienie

1. Polecenia import powodują przyłączenie potrzebnego modułu. W tym wypadku numpy – obliczenia numeryczne oraz csv – aby odczytać dane.

2. Konstrukcja **with open(lista.csv) as csvfile**: \; **readCSV = csv.reader(csvfile, delimiter=;)** powoduje utworzenie obiektu zwracającego kolejne wiersze odczytane z pliku **lista.csv**. Można jej używać bez zagłębiania się w szczegóły (jedyna zmiana wobec standardu skopiowanego z internetu to nazwa pliku z danymi. Jednak dla kompletności wykładu należy dodać, że:

a. Tak zwany delimiter to nic innego jak znak rozdzielającego pola w pliku csv. Jeśli zechcemy zapisać dane z arkusza kalkulacyjnego do tego typu pliku – musimy zdecydować jaki to ma być znak. Jeśli wybierzemy średnik, to typowy wiersz pliku będzie wyglądał następująco: Natalia;Nowak;8

b. Słowo kluczowe **with** pojawiło się całkiem niedawno (w Pythonie 2.6) i znaczy ono jedynie tyle, że mamy czytać z pliku jeśli ten plik istnieje i uda się go otworzyć (bardziej dogłębne wyjaśnienie: http://users.uj.edu.pl/~ufkapano/algorytmy/lekcja07/with.html).

3. Instrukcja for – z którą już zetknęliśmy się (w przykładzie 1.3) wykonuje działania na wszystkich elementach. Zapis **for row in readCSV**: oznacza pobieranie z obiektu odczytującego dane (readCSV) kolejnych wierszy (od angielskiego row). Elementy tego wiersza są numerowane od 0. Mamy więc 0=>Imię, 1=> Nazwisko, 2=>wiek. Zapis **row[2]** oznacza zatem wiek.

4. Zmienna ages (od angielskiego age=wiek) zawiera listę liczb. Inicjuje się ją wstawiając pustą listę oznaczoną []. Dodawaniu do listy służy własność append (ages.append(7) dodaje liczbę 7 do listy). Przygotowując ten przykład

5. Autor nie wziął pod uwagę tego, że zmienna row[] zawiera kolejne elementy wiersza pliku w postaci napisów. Uzyskał więc nie listę liczb, ale listę łańcuchów znaków (liczb w postaci napisów). To spowodowało pojawienie się tajemniczego błędu zgłaszanego przez funkcję median. Aby tego błędu uniknąć trzeba było zastosować tak zwaną konwersję typów. Zapis **int(9)** oznacza liczbę całkowitą samo **9** to tylko napis. Nie można na przykład napisać w programie '9'-9. Można natomiast 9-int('9'). Usunięcie wspomnianego błędu zajęło mniej więcej tyle czasu, co napisanie programu. Warto zdawać sobie sprawę z tego, że tak właśnie wygląda praca programisty.

Powyższy przykład został opisany dość dokładnie. Warto jednak poświęcić trochę czasu na eksperymenty z nim. Jeśli ktoś dobrze go zrozumie – może śmiało uważać siebie za programistę ;-).

Ćwiczenia do samodzielnego wykonania

Dokonaj następujących modyfikacji programu:

1. Zafałszuj dane wstawiając dodatkowe elementy do listy (na przykład ages.append(7)). Zwróć uwagę na to jak zmienia się mediana.

2. Sprawdź działanie programu posługując się poznaną w przykładzie 1.2 metodą sort (ages.sort()) i poleceniem print.

Instrukcja for

Instrukcja for (`for . . in`) jest zaliczana do tak zwanych pętli – czyli powtarzania działań tak długo, aż pojawią się warunki zakończenia pętli (na przykład nie ma więcej danych do przetworzenia). Chcąc przedstawić coś w miarę prostego w skomplikowany sposób, można napisać, że pętla **for** definiuje *iteracje ** nad sekwencją danych*. Nie ma jednak w tym nic tajemniczego lub trudnego. Chodzi wyłącznie o to, że kolejne dane są pobierane do przetwarzania w określonym porządku (w przykładzie 3.1 w kolejności zapisania w pliku), a działania zostaną wykonane na wszystkich danych (chyba, że nastąpi błąd przerywający program).

Jeśli ktoś zetknął się z programami w innych językach, to może mu się nasunąć w tym miejscu pytanie: a co zrobić, jeśli nie chcę wykonać instrukcji zdefiniowanych w pętli dla wszystkich danych, ale określoną ilość razy? Ale przecież to jest to samo! Po prostu dane jakie należy użyć w pętli (po słowie kluczowym **in**) są kolejnymi liczbami! W Pythonie zapisujemy sekwencję liczb w postaci: **range(od,do)**. Na przykład **range(1,5)** oznacza 4 liczby kolejne od 1 do 4. Czemu do 4? Bo drugi parametr w range oznacza górne ograniczenie (czyli dostajemy w tym przypadku liczby mniejsze od 5).

Przykład 3.2: **i_for**

```
1 for i in range(1, 5):
2   print(i)
3 else:
4   print('Zrobione')
```

Wyjście:

```
1
2
3
4
Zrobione
```

Objaśnienie

1. Wbudowana funkcja range działa jak generator kolejnych liczb (od 1 do 4). Możemy też wygenerować ciąg arytmetyczny inny niż złożony z kolejnych liczb. Funkcja range akceptuje bowiem trzeci parametr określający krok – co ile zwiększamy kolejne liczby. Na przykład, `range(1,5,2)` daje tylko dwie liczby `[1,3]` . Pamiętaj, że drugi parametr (zakres) określa górne ograniczenie, a nie ilość liczb!

2. Inną ważną kwestią jest to, że `range()` jest generatorem zwracającym kolejne liczby pojedynczo, a nie wszystkie liczby naraz. Nie można więc zapisać: lista=range(1,5). Jeśli chcemy uzyskać listę pięciu kolejnych numerów, należy użyć konstrukcji `lista=list(range(1,5))`.

3. Funkcja range może być wywołana z jednym parametrem. Wtedy przyjmuje się, że dolne ograniczenie wynosi 0. Zapis `list(range(5))` daje nam liestę `[0, 1, 2, 3, 4]`.

4. Pętla for posiada element opcjonalny oznaczony słowem kluczowym else (inaczej). Do tego miejsca przechodzi program po wykonaniu pętli (lub jeśli nie ma nic do wykonania).

Ćwiczenie do samodzielnego wykonania: 1. Modyfikuj program zmieniając parametry w funkcji range. Obserwuj co się zmieni, gdy podamy na przykład range(3), range(3,8), range(0,10,2). 2. A co się zmieni, jeśli zamiast range(1,5) wpiszemy list(range(1,5))? Podpowiedź: lista też może być użyta w pętli for jako źródło (generator) danych.

Struktury danych

We wcześniejszych przykładach zostały użyte bez specjalnego wprowadzenia listy danych. Są to przykłady struktur danych, które mogą być przechowywane w zmiennych. Zamiast jednej prostej danej (jak liczba czy napis) w jednej zmiennej może być wiele powiązanych danych. Poza listami istnieją w Pythonie trzy inne rodzaje struktur: krotka (inaczej kolekcja, rekord), słownik (inaczej tablica asocjacyjna) i zbiór. Struktury te różnią się sposobem w jaki możemy sięgać do elementów (danych) zawartych w nich.

Sposób zapisu oraz podstawowy sposób dostępu do danych zawiera poniższa tabelka:

Struktura	Identyfikator	Przykład	Typowe użycie
lista	list	Liczby = [1,4,6,3]	Operacje na listach danych (sortowanie, łączenie i dzielenie, modyfikacje)

Struktura	Identyfikator	Przykład	Typowe użycie
krotka	tuple	Czerwień = (255, 0, 0)	Rekord bazy danych, parametr. Krotek nie można modyfikować!
słownik	dict	Numery = {'raz': 1, 'dwa': 2, 'trzy': 3}	Dostęp do danych poprzez nazwę.
zbiór	set	set(['Gniezno', 'Kraków', 'Warszawa'])	Sprawdzenie czy jest w zbiorze

Jak widać struktury różnią się w zapisie rodzajem użytych nawiasów (zbiory mogą być tworzone jedynie przez funkcję – a nie poprzez samo użycie nawiasów. Lepsze zrozumienie zastosowania tych struktur uzyskamy analizując poniższe przykłady.

Listy

Lista służy do przechowywania uporządkowanej kolekcji obiektów (danych). Przykładem takiej listy z którą na pewno każdy się zetknął jest lista zakupów. Praktycznie wszystkie dane można zapisać w postaci list. Inne struktury stosuje się dla wygody (na przykład aby ograniczyć bezsensowne próby modyfikacji). W Pythonie elementy listy dzieli się przecinkami. Całość obejmują nawiasy kwadratowe. Każdy element listy ma swój numer lista_dzieci[5] określa piąty element listy o nazwie lista_dzieci.

Przykład 3.3: **using_list**

```
# Lista zakupów
shoplist = ['jabłka', 'mango', 'marchewka', 'banany']
print(u'Mam %s rzeczy do kupienia. \nSą to:' % len(shoplist))
for item in shoplist:
  print(item),
# w Pythinie 3.x:    print(item, end=' ')
print(u'\nDla sąsiadki kupuję jeszcze ryż.')
shoplist.append('rice')
print(u'pełna lista zakupów to obecnie %s' % shoplist)

print('Posortowana lista:')
shoplist.sort()
print(shoplist)

print(u'Skreślam pierwszy element, bo mam mało pieniędzy: %s' % shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print(u'Zapamiętam, że nie kupiłem: %s' % olditem)
print(u'Ostateczna lista zakupów: %s' % shoplist)
```

Wyjście:

```
Mam 4 rzeczy do kupienia.
Są to:
jabłka mango marchewka banany
Dla sąsiadki kupuję jeszcze ryż.
pełna lista zakupów to obecnie ['jab\xc5\x82ka', 'mango', 'marchewka', 'banany',\
 'rice']
Posortowana lista:
['banany', 'jab\xc5\x82ka', 'mango', 'marchewka', 'rice']
Skreślam pierwszy element, bo mam mało pieniędzy: banany
Zapamiętam, że nie kupiłem: banany
Ostateczna lista zakupów: ['jab\xc5\x82ka', 'mango', 'marchewka', 'rice']
```

Objaśnienie

1. Zmienna `shoplist` jest listą zakupów (nazw rzeczy). Lista może być użyta jako źródło sekwencji danych w pętli **for**.

2. W Pythonie 3.x wprowadzono parametr **end** do funkcji `print`. Zmienia on sposób zakończenia druku/wyświetlania danych. Zamiast normalnego przejścia do nowej linii, pojawia się znak określony w parametrze end. Wbrew pozorom różnice między podobnie wyglądającym zapisem w wywołaniu print() w Pythonie 2.x i 3.x są bardziej znaczące (w Pythonie 2.x print (1,2,3) oznacza wydruk krotki (1,2,3)). Zapis zastosowany w powyższym przykładzie jest uniwersalny (działa w każdej wersji Pythona)/

3. Gdyby funkcja print dopuszczała jedynie dwa parametry – można by skrócić zapis `end=` do ‘ ‘. Jednak w print może być więcej parametrów i musimy wskazać przez nazwę do którego odnosi się nasze odwołanie.

4. Poza poznanymi we wcześniejszych przykładach metodami działania na liście (dodanie: `append`, sortowanie `sort`), użyto usuwania z listy poleceniem `del`. Instrukcja `del shoplist[0]` usuwa pierwszy element z listy (należy pamiętać, że Python rozpoczyna liczenie od 0). Wśród metod obiektu listy nie ma odpowiednika **del**. Do usuwania może służyć metoda remove. Wymaga ona jednak podania w parametrach usuwanego obiektu, a nie jego numeru. W tym wypadku: shoplist.remove(banany). Zewnętrzną funkcją jest też **len()** określającea długość listy.

Krotki

Krotki są używane do przechowywania razem kilku obiektów. Są one podobne do list, ale pozbawione części funkcjonalności (nie można ich modyfikować). Krotki są definiowane poprzez umieszczenie elementów rozdzielonych przecinkami w nawiasach okrągłych () - podczas gdy listy określa się nawiasami kwadratowymi.

Krotki są zwykle stosowane w przypadkach, gdy chcemy uniknąć możliwości zmieniania danych. Na przykład dane odczytane z bazy danych powinny być używane w niezmienionej formie.

Przykład 3.3: **using_tuple**

```
1  zoo = ('python', 'elephant', 'penguin')
2  print('Ilość zwierząt w ZOO =%s' % len(zoo))
3  new_zoo = 'monkey', 'camel', zoo
4  print(u'W nowym ZOO są zwierzęta:')
5  print(new_zoo)
6  print(u'razem %s miejsca' %  len(new_zoo))
7  print(u'Zwierzęta nabyte ze starego zoo:')
8  print(new_zoo[2])
9  print(u'Ostatnie zwierzę ze starego zoo to %s' % new_zoo[2][2])
10 print u'Ilość zwierząt w nowym zoo: ', len(new_zoo)-1+len(new_zoo[2])
```

Wyjście:

```
Ilość zwierząt w ZOO =3
W nowym ZOO są zwierzęta:
('monkey', 'camel', ('python', 'elephant', 'penguin'))
razem 3 miejsca
Zwierzęta nabyte ze starego zoo:
('python', 'elephant', 'penguin')
Ostatnie zwierzę ze starego zoo to penguin
Ilość zwierząt w nowym zoo:  5
```

Objaśnienie

1. Zmienna `zoo` zawiera elementy krotki. Widzimy, że funkcja `len` podobnie jak w przypadku list zwraca ilość elementów. Podobnie jak listy – krotki mogą być użyte jako sekwencja przeglądana w pętli for.

2. Po likwidacji starego ZOO zwierzęta zostały przeniesione do oddzielnej lokalizacji w nowym ZOO. Zainicjowanie zmiennej `new_zoo` twory nową krotkę zawierającą dwa nowe zwierzęta, a w trzecim elemencie krotki – dane o zwierzętach ze starego ZOO. Ta operacja wymaga kilku słów komentarza:

- Pominięto w niej nawiasy – gdyż one są przy tworzeniu krotek opcjonalne (niemniej zachęca się do ich stosowania) wyrażenie a=1,2,3,4 jest równoważne a=(1,2,3,4) – można się o tym przekonać wykonując druk (print) a lub a.__class__.
- Możliwość umieszczania struktury w strukturze pozwala tworzyć listy list (zamiast tablic wielowymiarowych) i inne złożone struktury. Funkcja len liczy ilość elementów struktury podanej w parametrze – a wewnętrzne struktury są traktowane jak jeden element. Dlatego len(new_zoo) zwraca 3 – choć ilość zwierząt jest większa.

3. Podobnie jak w przypadku list, możemy uzyskać dostęp do elementów w krotce określając pozycję elementu numerem podanym w parze nawiasów kwadratowych. Jest to tak zwany operator *indeksowania* ** Trzeci elementu w `new_zoo` ma indeks `2` (liczymy od `0`) i możemy odwołać się do niego pisząc `new_zoo[2]`. Trzecia pozycja w tym trzecim elemencie (zwierzęta ze starego ZOO) to `new_zoo[2][2]`.

UWAGA Pusta krotka jest zapisywana w postaci samych nawiasów: `myempty = ()`. Jednak należy uważać na krotki z jednym elementem. Bo jak Python ma odróżnić wyrażenie w nawiasach od krotki? Aby rozwiązać ten problem, stosuje się przecinek po ostatnim (w ty wypadku jedynym) elemencie krotki: `singleton = (2, )`. Bez przecinka (2) będzie traktowane jak liczba (możesz sprawdzić drukując (2).__class__.

Każde dwa elementy rozdzielone przecinkiem (lub jedna zakończona przecinkiem) są traktowane jak krotka.

```
1   a=1,
2   krotka=1,a,3
3   print(krotka)
```

wynik:

```
(1, (1,), 3)
```

Słowniki

Słownik można przyrównać do książki adresowej, gdzie można znaleźć adres lub dane teleadresowe osoby, znając tylko jego / jej imię i nazwisko. Słownik kojarzy *klucze* (nazwy) z *wartościami* (szczegóły). Stąd używana czasem nazwa 'tablice asocjacyjne' (od asocjacja = skojarzenie). Należy pamiętać, że klucz musi być unikalny. Gdyby rzeczywistą książkę adresową chcieć umieścić w takiej strukturze, to pojawi się problem z różnymi osobami o tym samym nazwisku. Nie będzie takiego problemu, jeśli posłużymy się numerem PESEL.

Klucze w słowniku są niezmienne. Zmieniać można natomiast wartości. Czyli klucze powinny być danymi prostymi (liczby, napisy). Słowniki zapisujemy w postaci par klucz:wartość, rozdzielonych przecinkami i ujętymi w nawiasy klamrowe. Na przykład: `d = {k`lucz1 : wartosc1, klucz2 : wartosc2 }'`.

Pary klucz-wartość w słowniku nie są uporządkowane w żaden sposób. Słownik nie może więc być przeglądany w pętli for (jako sekwencja użyta po słowie in). Jednak w obiekcie słownika mamy trzy metody zwane widokami, które zwracają odpowiednio listę par, listę kluczy i listę wartości: dict.keys(), dict.values(),dict.items(). Słownik **{1:jeden,2:dwa}** jest klasy dict, ale **{1:jeden,2:dwa}.items()** to już lista.

Przykład 3.5: **using_dict**

```
# 'ab' to krótka lista adresowa
ab = {
'Swaroop': 'swaroop@swaroopch.com',
'Larry': 'larry@wall.org',
'Matsumoto': 'matz@ruby-lang.org',
'Spammer': 'spammer@hotmail.com'
}
print("Adres Swaroop'a to", ab['Swaroop'])
# Usuwanie pary z listy
del ab['Spammer']
print('Ilość kontaktów w liście: {}\n'.format(len(ab)))
for name, address in ab.items():
  print('Adres {} to {}'.format(name, address))
# dodanie pary
ab['Guido'] = 'guido@python.org'
if 'Guido' in ab:
  print("\nAdres Guido'to", ab['Guido'])
```

Objaśnienie

1. Tworzymy słownik ab pomocą wcześniej omówionej notacji. Uzyskujemy dostęp do adresów na podstawie nazwiska – stosując prostą notację z nawiasami klamrowymi. Można też używać usuwania (del) – tak jak w listach.

2. W programie pokazano przeglądanie elementów słownika z wykorzystaniem metody items(). Zwraca ona po kolei pary klucz, wartość – jako elementy krotki.

3. Dodawanie i zmiana elementów słownika wygląda tak samo. Zapis: ab[Guido] = guido@python.org[22] oznacza, że jeśli nie ma danych dla klucza 'Guido' zostanie dodany do słownika element z takim kluczem. Jeśli już w słowniku jest – nastąpi zmiana adresu.

4. Sprawdzenie, czy element o danym kluczu jest na liście, wykonujemy operatorem ** in** (zobacz zapis **if Guido in ab**:).

Zbiory

Zestawy to *nieuporządkowan**a* kolekcja prostych obiektów. Są one stosowane, gdy ważne jest tylko istnienie obiektu w kolekcji, lub ilość wystąpień. Można je kojarzyć ze zbiorami w matematyce. Mamy podobne operacje: sprawdzenie członkostwa, podzbiory, przecięcia itd.…

[22] mailto:'guido@python.org

bri = set([‘brazil’, ‘russia’, ‘india’]) print(‘india’ in bri) bric = bri.copy() bric.add(‘china’) print(‘BRI=’); print(bri) print(‘BRIc=’);print(bric) print(‘BRI podzbiorem BRIc?’) print(bric.issuperset(bri)) bri.remove(‘russia’) print(bric) print(bri & bric) bric.intersection(bri) print(bric)

Wyjście:

```
True
BRI=
set(['brazil', 'india', 'russia'])
BRIc=
set(['brazil', 'china', 'india', 'russia'])
BRI podzbiorem BRIc?
True
set(['brazil', 'china', 'india', 'russia'])
set(['brazil', 'india'])
set(['brazil', 'china', 'india', 'russia'])
```

Objaśnienie

Przykład jest dość prosty, ponieważ wiąże się podstawami matematyki teorii mnogości nauczanych w szkole. Zwróć uwagę na możliwości umieszczania kilku instrukcji w jednym wierszu, rozdzielonych przecinkami.

Python jako język zapisu algorytmów

Na początku podręcznika podkreślono, że programując w Pythonie staramy się opisać co ma być zrobione, a nie w jaki sposób. Czasem jednak nie udaje się uniknąć implementowania algorytmów (czyli szczegółowych przepisów - jak coś zrobić). W takim przypadku operowanie na danych i użycie pętli **for** nie wystarczy. Konieczne jest użycie instrukcji sterowania oraz manipulowania danymi. Następny będzie poświęcony manipulowaniu danymi. W tym rozdziale poznamy natomiast bliżej instrukcje sterowania kolejnością wykonywanych instrukcji. Więcej informacji na ten temat znajdziesz w modułowym kursie programowania[23] - z moduł Programy_a_algorytmy[24] a zwłaszcza z podrozdział Implementacja_algorytmów[25].

Instrukcje programów w Pythonie są wykonywane sekwencyjnie – w kolejności zapisania. Jednak algorytmy które implementujemy w programach wymagają powtarzania pewnych operacji lub wykonania ich jedynie pod pewnymi warunkami. Powtarzanie fragmentu kodu zapisuje się instrukcją pętli a opcjonalne wykonanie fragmentu kodu - instrukcją warunkową.

Dotąd zetknęliśmy się z instrukcją pętli for – przy pomocy której zapisywane były programy przetwarzania struktur danych. W niniejszym rozdziale poznamy instrukcję warunkową (**if**) oraz instrukcję pętli (**while**). Pętla **while** różni się od pętli **for** przede wszystkim tym, że nie musimy mieć w chwili jej rozpoczęcia informacji o zbiorze danych jaki będzie przetwarzany. Warunkiem powtarzania pętli może być dowolne wyrażenie logiczne (tak zwany niezmiennik pętli).

Spośród spotykanych w innych językach instrukcji w Pythonie nie ma **case** oraz **repeat**.

Być może dla kogoś ułatwieniem w zrozumieniu działania instrukcji będzie zabawa z klockami, jaka jest dostępna na stronie: http://www.otwartaedukacja.pl/programowanie/bl/code/

Zawsze gdy w opisach jest mowa o blokach (blokach kodu) – należy pamiętać o tym, że wyróżnia się je wcięciami – czyli ilością spacji z przodu (po lewej stronie) wiersza.

Instrukcja warunkowa (if)

Instrukcja **if** służy do zaznaczenia bloku instrukcji, który się wykona wyłącznie wtedy, gdy prawdziwe będzie wyrażeni logiczne sprawdzane przed wejściem do tego bloku. Jeśli warunek jest spełniony, wykonywany jest blok instrukcji (możemy go określić jako “blok warunkowy”). w przeciwnym wypadku możemy wskazać inny blok instrukcji do wykonania. Służy do tego słowo kluczowe **else**. Użycie **else** jest opcjonalne.

[23] http://otwartaedukacja.pl/programowanie
[24] http://otwartaedukacja.pl/programowanie/index.php?title=Programy_a_algorytmy
[25] http://otwartaedukacja.pl/programowanie/index.php?title=Implementacja_algorytmów

Przykład 4.1: **ex_if**

```
number = 23
guess = int(input('Podaj liczbę całkowitą: '))
if guess == number:
  # Blok warunkowy zaczyna się w tym miejscu
  print('Gartulacje, zgadłeś liczbę.')
  print('(ale niestety nie ma żadnej nagrody!)')
  # Tu kończy się blok warunkowy
elif guess < number:
  # Drugi blok
  print('Nie, liczba powinna być większa')
  # Koniec drugiego bloku ...
else:
  print('Nie, liczba powinna być mniejsza')
  # ten blok zostanie wykonany tylko gdy guessed > number
print('Koniec')
# Ostatnia instrukcja (print) jest wykonywana zawsze - po zakończeniu instrukcji\
 warunkowej.
```

Wyjście:

```
Podaj liczbę całkowitą: 25
Nie, liczba powinna być mniejsza
Koniec
```

Objaśnienie

Program pozwala na wprowadzanie liczby od użytkownika (funkcją **input()**) i sprawdzanie, czy zgadł on liczbę zapisana w pamięci zmienna number). Następnie w zależności od tego, czy wprowadzono liczbę zapamiętaną (23), mniejszą lub większą – wykona si inny blok instrukcji.

Funkcja wbudowana **input** drukuje na ekranie parametr i czeka na dane zakończone klawiszem Enter. Następnie **input()** zwraca wprowadzony ciąg znaków.

W programie ciąg znakow jest konwertowany do liczby (za pomocą **int)** i zapisywany do zmiennej **guess** (zgadywanie).

Sprawdzane wyrażenia logiczne to porównywanie liczb zapamiętanych w zmiennych numer i guess.

Bloki instrukcji są zaznaczane poprzez wcięcia (spacje z lewej).

Zauważ, że **if** sprawozdanie zawiera dwukropek na końcu - jesteśmy wskazując Pythonie, że blok sprawozdania następuje.

W programie użyto słowa kluczowego **elif**. Jest to skrót od dwóch słów: **else-if** i działa właśnie tak – jako ich połączenie. Dzięki temu zmniejsza się ilość wymaganych wcięć (porównaj ten sam tekst po zamianie **elif** na **else**: **if** ….).

Zarówno **elif** jak i **else** są opcjonalne. Minimalna struktura instrukcji **if** może wyglądać następująco:

```
1  if True:
2     print('Tak, to prawda')
```

Co najmniej jedna instrukcja w bloku warunkowym jest wymagana!!! Jeśli nic nie ma do zrobienia – użyj instrukcji pustej **pass**.

Pęlta while

Pętla **while** jest podobna do instrukcji warunkowej (**if**). W obu wypadkach blok instrukcji wykonuje się pod warunkiem, że podany warunek (wyrażenie logiczne) jest spełniony. Jedyna różnica polega na tym, że blok warunkowy w instrukcji **if** wykona się raz, gdy tymczasem blok pętli **while** wykonuje się tak długo, jak długo warunek jest spełniony.

Złóżmy, że mamy zmienną **n** w której aktualnie jest liczba 5. Porównajmy dwa przykłady:

1)

```
1  if n>0:
2      print(n)
3  n=n-1
```

oraz:

2)

```
1  while n>0:
2    print(n)
3    n=n-1
```

W przykładzie 1) zostanie wyświetlona liczba 5, a po zakończeniu instrukcji wartość n wynosi 4. W przypadku 2) zostaną wypisane liczby od 5 do 1, a po skończeniu instrukcji n wyniesie 0.

Zmodyfikujmy analogicznie przykład ze zgadywaniem liczb – zezwalając na wielokrotne odgadywanie poprzez zamianę **if** na **while**.

***Przykład** 4.2:* **ex_while**

```
number = 23
running = True
while running:
  guess = int(input('Podaj liczbę całkowitą: '))
  if guess == number:
    # Blok warunkowy zaczyna się w tym miejscu
    print('Gartulacje, zgadłeś liczbę.')
    print('(ale niestety nie ma żadnej nagrody!)')
    # Tu kończy się blok warunkowy. Konczy se także petla!
    running = False
  elif guess < number:
    # Drugi blok
    print('Nie, liczba powinna być większa')
    # Koniec drugiego bloku ...
  else:
    print('Nie, liczba powinna być mniejsza')
    # ten blok zostanie wykonany tylko gdy guessed > number
else:
  print('Koniec')
```

Wyjście:

```
Podaj liczbę całkowitą: 26
Nie, liczba powinna być mniejsza
Podaj liczbę całkowitą: 21
Nie, liczba powinna być większa
Podaj liczbę całkowitą: 25
Nie, liczba powinna być mniejsza
Podaj liczbę całkowitą: 22
Nie, liczba powinna być większa
Podaj liczbę całkowitą: 23
Gartulacje, zgadłeś liczbę.
(ale niestety nie ma żadnej nagrody!)
Koniec
```

Objaśnienie

Zabawa w zgadywanie prawie taka sama jak poprzednio – ale powtarzamy ją w pętli dopóki użytkownik nie zgadnie. Wskazuje na to zmienna logiczna **running**. To proste wyrażenie logiczne (z jednej zmiennej) jest zawsze prawdziwe w obrębie pętli. Dlatego nazywa się to niekiedy niezmiennikiem pętli. Należy zwrócić szczególną uwagę na to, że jeśli nie zmienimy tego wyrażenia – petla

może nigdy się nie skończyć. Dlatego należy zwrócić szczególną uwagę na to, żeby niezależnie od tego jakie dane będą przetwarzane – nastąpiły zmiany powodujące wyjście z pętli. Jest to jdna z przyczyn dla których uzycie pętli **for** jest prostsze.

Blok **else** jest wykonywany, gdy **while** warunek (niezmiennik) pętli staje **False** - może to nastąpić nawet gdy warunek nie jest nigdy spełniony (wtedy działa to dokładnie jak if).

W Pythonie istnieje dwie dodatkowe instrukcje pozwalające sterowaniem pętli: **continue** i **break** (zob. dalej). Nie są one w pełni zgodne z ideą programowania strukturalnego – ale bywają wygodne. Nie należy ich nadużywać, ale warto znać.

Instrukcje break i continue

Instrukcja **continue** powoduje pominięcie reszty bloku instrukcji w pętli. Nie ma ona więc sensu, gdy występuje poza pętlą, lub jako ostatnia instrukcja w bloku. Po wykonaniu instrukcji Python przystępuje do wykonania następnej iteracji (powtórzenia) pętli. Może

***Przykład** 4.3:* **ex_continue**

```
1  points = 7 # pierwsza liczba jest ukryta
2  counter = points # suma punktow
3  print('Możesz zakończyć sumowanie wpisując 0')
4  while counter < 21 and points != 0:
5    points = input('Wpisz liczbę od 0 do 12: ')
6    if points<0 or points>12:
7      print('Od 0 do 12!')
8      continue
9    counter+= points # dodanie liczby
10 print('Suma=%s' % counter)
```

Przykładowe wyjście

```
Możesz zakończyć sumowanie wpisując 0
Wpisz liczbę od 0 do 12: 14
Od 0 do 12!
Wpisz liczbę od 0 do 12: 13
Od 0 do 12!
Wpisz liczbę od 0 do 12: 12
Wpisz liczbę od 0 do 12: 0
Suma=19
```

Objaśnienie

1. To gra w zgadywanie – podobna do pewnej gry karcianej. Ukryta jest pierwsza liczba punktów (points). Podajemy następne uważając, by nie przekroczyć 21. Liczby muszą być od 1 do 12. Dodatkowo 0 kończy grę(gdy grający boi się przegrać i przekroczyć 21). Instrukcja continue jest tu wykorzystana do tego, aby zignorować liczby nie spełniające warunków (0..12).

2. Należy pamiętać o tym, że **continue** działa także w pętli **for**.

Powyższy przykład zawiera dwa typowe elementy: ignorowanie błędnych danych w pętli je obrabiającej oraz wykorzystanie dodatkowej zmiennej dla zmiany wyrażenia pętli (niezmiennika).

Ten drugi element nieco zaciemnia czystość kodu. Gramy o to, by uzyskać sumę mniejszą od 21, ale nie przekroczyć tej liczby. Dobrze byłoby, aby warunek pętli odnosił się do rzeczywistości, a nie był sztuczny. Można do tego wykorzystać instrukcję **break**, która służy do *przerwania* pętli bez względu na to czy zostały spełnione warunki jej zakończenia. Program przechodzi do następnej instrukcji po pętli. Działa zarówno w pętli **for** jak i **while**.

***Przykład** 4.3:* **ex_break**

```
points = 4 # pierwsza liczba jest ukryta
counter = points # suma punktow
print('Możesz zakończyć sumowanie wpisując 0')
while counter < 21:
  points = input('Wpisz liczbę od 0 do 12: ')
  if points<0 or points>12:
    print('Od 0 do 12!')
    continue
  if points==0: # kończymy grę
    break
  counter += points # dodanie liczby
  if counter>21: # przegrana
    break
print('Suma=%s' % counter)
```

Przykładowe wyjście:

```
Możesz zakończyć sumowanie wpisując 0
Wpisz liczbę od 0 do 12: 13
Od 0 do 12!
Wpisz liczbę od 0 do 12: 12
Wpisz liczbę od 0 do 12: 7
Suma=23
```

Objaśnienie

Blok instrukcji w pętli zostaje przerwany, gdy uzyskamy dane czynią bezsensownym dalsze ich przetwarzanie. Może to dotyczyć danych istotnych tylko w tej iteracji – wtedy stosujemy **continue** (dane nie spełniają warunków), albo całej pętli (wychodzimy poza niezmiennik pętli) – wtedy przerywamy (**break**). K każdym razie – zarówna dla **continue** jak i **break** mamy pominięcie reszty bloku, ale **break** dodatkowo powoduje zakończenie całej pętli.

Pamiętaj, że **break** można stosować także w pętli **for**.

Podsumowanie

Poznane instrukcje wyczerpują schematy potrzebne do zapisania dowolnych algorytmów.

Jak radzić sobie ze złożonością

Wielkość programów liczonych w wierszach kodu rzadko jest mniejsza niż tysiąc. Czasami jest liczona w milionach. Dlatego umiejętność radzenia sobie z ich złożonością jest w pracy programisty ważniejsza niż opanowanie języka programowania. W tym rozdziale poznamy mechanizmy używane do budowania złożonych programów w języku Python.

Funkcje

W większości podręczników programowania funkcje definiuje się jako fragmenty kodu wielokrotnego stosowania. Na przykład w kodzie (https://pl.wikipedia.org/wiki/Szyfr_Solitaire[26]):

```
1  if liczba < 0:
2    liczba = 256+liczba
3    szyfr = (liczba + klucz) % 256
4  else:
5    szyfr = (liczba + klucz) % 256
```

Powtarza się wyliczenie wartości zmiennej **szyfr**. Możemy wydzielić ten fragment nadając mu jakąś nazwę (tu: szyfruj):

```
1  def szyfruj():
2    szyfr = (liczba + klucz) % 256
3
4  # przykład wymaga poprawienia !!!!! – zob. opis
5  if liczba < 0:
6    liczba = 256+liczba
7    szyfruj()
8  else:
9    szyfruj()
```

Słowo kluczowe **def** oznacza definicję funkcji – czyli właśnie tego powtarzalnego fragmentu programu. Po nagłówku zaczynającym się od def następuje blok kodu stanowiący tak zwane „ciało" funkcji. Interpreter Pythona napotykając napis szyfruj() odszukuje definicję funkcji i wykonuje blok ciała funkcji.

[26] https://pl.wikipedia.org/wiki/Szyfr_Solitaire

W takim prostym stosunkowo kodzie korzyści zastosowania funkcji nie są oczywiste - zwłaszcza, że rodzi ono dodatkowe problemy o których trzeba pamiętać.

Przykład powyższy nie jest w pełni poprawny. Zmienne są tworzone z chwilą ich użycia. Jeśli zmienna pojawi się w funkcji – to jest nazywana zmienną lokalną i nie istnieje po zakończeniu funkcji. Jeśli więc dopiszemy na końcu print(szyfr) to nie wydrukuje się wynik działania funkcji. Poniżej znajduje się wyjaśnienie pojęcia wyniku funkcji, dzięki któremu można ten program poprawić.

Na dodatek łatwo zmienić kod programu eliminując powtórzenia:

```
1 if liczba < 0:
2     liczba = 256+liczba
3 szyfr = (liczba + klucz) % 256
```

Ten sposób upraszczania kodu jest lepszy niż stosowanie funkcji. Podstawowym zastosowaniem funkcji nie jest wcale eliminacja powtórzeń, ale właśnie uczynienie kodu bardziej czytelnym. Jeśli algorytm szyfrowania jest złożony, albo chcemy zwrócić uwagę na to, że fragment kodu jest definicją szyfrowania – stosujemy funkcję – nawet gdy w danym momencie nie przewidujemy powtórzeń:

```
1 if liczba < 0:
2     liczba = 256+liczba
3 szyfruj()
```

Określenie funkcji jako wydzielenie powtarzanego fragmentu kodu nie jest dobre jeszcze z jednego powodu: sugeruje, że możemy po prostu zrobić kopiuj i wklej powtarzalnego fragmentu. To nie jest do końca prawda. Funkcja powinna być kodem autonomicznym – czyli jak najmniej powiązanym z innymi fragmentami. Gdzie są używane w niej zmienne liczba, klucz i szyfr? W dalszych rozdziałach podręcznika poznamy działanie funkcji bardziej dogłębnie, ale na początek musimy przyswoić sobie pojęcie **parametru** i **wyniku funkcji**. Parametrem jest zmienna której wartość przekazywana jest w chwili wywołania (użycia) funkcji. Wynikiem jest wartość przekazywana na zewnątrz. Służy do tego polecenie **return**.

```
1 def szyfruj(liczba, klucz):
2     return (liczba + klucz) % 256
```

W powyższym przykładzie zdefiniowaliśmy parametry (liczba i klucz) – do tego właśnie służą nawiasy po nazwie funkcji. A co się stało ze zmienną zawierającą wynik (szyfr)? Proszę zwrócić uwagę na to, że powyżej zdefiniowaliśmy wynik jako **wartość** przekazywana na zewnątrz. A więc wartość zmiennej, a nie zmienną. Do jakiej zmiennej wstawić tą wartość definiujemy w chwili użycia (wywołania) funkcji:

```
szyfr = szyfruj(liczba, klucz)
```

Warto zwrócić uwagę na konwencję zastosowaną w nazwach. Nazwy zmiennych i parametrów określono rzeczownikami (opisują one **co** zmienna przechowuje), a funkcji czasownikiem (jakie działania należy wykonać na danych). Taka konwencja nie jest obowiązkowa i nie jest stosowana konsekwentnie – ale ułatwia ona czytanie kodu.

Autonomiczność funkcji wiąże się także z tym, że nazwy parametrów są istotne tylko wewnątrz funkcji. Nazwy przekazywanych zmiennych mogą być zupełnie inne. W miejsce zmiennych można zresztą używać w wywołaniu funkcji dowolnych wyrażeń. Przekazywana jest bowiem wartość, a nie zmienna. A oto kompletny przykład:

Przykład 5.1: **cipher0**

```
#encoding: UTF-8

def szyfruj(liczba, klucz):
    return (liczba + klucz) % 256

def odszyfruj(liczba, klucz):
    return (256 + liczba - klucz) % 256

wiadomosc = int(input('Podaj dodatnią liczbę do zaszyfrowania: '))
szyfr=szyfruj(wiadomosc, 89)
print('Wynik=%s' % szyfr)
print('Odszyfrowany: %s' % odszyfruj(szyfr, 89))
```

Objaśnienie

Pobrany ciąg znaków zostaje zamieniony na liczbę (w stosunku do wcześniejszego opisu zrezygnowaliśmy z liczb ujemnych). Następnie zostaje wywołana funkcja szyfruj. Parametr liczba przyjmuje wartość zmiennej wiadomosc, a parametr klucz przyjmuje wprost podaną wartość 89. Wynik działania funkcji zostaje zapisany do zmiennej szyfr.

Obiekty

Aby zapewnić poprawne działanie funkcji, należy posługiwać się tym samym kluczem do szyfrowania i deszyfracji. Nic prostszego – wystarczy wpisywać stale ten sam klucz w parametrze wywołania. A jeśli zechcemy zmienić klucz – to musimy szukać każdego wywołania. Możemy łatwo coś ominąć lub się pomylić. Istnieje jednak mechanizm grupujący w jednym miejscu dane i funkcje na nich działające. Nazywa się to programowaniem obiektowym. Dzięki niemu możemy parametry ustalić tylko raz – z chwilą definiowania obiektu. Obiektem jest właśnie zgrupowanie danych i funkcji na nich działających. Definicję obiektów nazywamy klasą i określamy słowem kluczowym class:

```
#encoding: UTF-8

class TSzyfrant:

    klucz=0

    def szyfruj(self, liczba):
        return (liczba + self.klucz) % 256

    def odszyfruj(self, liczba):
        return (256 + liczba - self.klucz) % 256

szyfrant=TSzyfrant()
szyfrant.klucz=89
wiadomosc = int(input('Podaj dodatnią liczbę do zaszyfrowania: '))
szyfr=szyfrant.szyfruj(wiadomosc)
print('Wynik=%s' % szyfr)
print('Odszyfrowany: %s' % szyfrant.odszyfruj(szyfr))
```

Funkcje zawarte w obiekcie (zdefiniowane dla klasy) nazywamy metodami (metoda zmiany danych). Jako pierwszy parametr jest do tych funkcji przekazywany obiekt na rzecz którego funkcja jest wykonywana.

***Przykład** 5.2:* **cipher1o**

```
class klasa():
    def metoda(self):
        print(self.wlasnosc)

obiekt1=klasa()
obiekt1.wlasnosc=1
obiekt2=klasa()
obiekt2.wlasnosc=2
obiekt1.metoda()
obiekt2.metoda()
```

*** Wyjście:***

```
1
2
```

Objaśnienie

W tym przykładzie pokazano czym różni się klasa od obiektu. Klasa została zdefiniowana w pierwszych trzech linijkach. Później zostały utworzone dwa obiekty tej klasy. Następnie została ustawiona własność klasy. Zwrócić należy uwagę na to, że własności są tworzone dynamicznie - nie musimy tego zawierać w definicji klasy. Jednak dla przejrzystości kodu nie zaleca się takiego programowania.

Często własności incjuje się w funkcji specjalnej __init__ - nazywanej konstruktorem, która jest wykonywana z chwilą tworzenia obiektu.

***Przykład** 5.3:* **cipher2o**

```
1  #encoding: UTF-8
2
3  class TSzyfrant:
4
5      def __init__(self,klucz):
6          self.klucz=klucz
7
8      def szyfruj(self, liczba):
9          return (liczba + self.klucz) % 256
10
11     def odszyfruj(self, liczba):
12         return (256 + liczba - self.klucz) % 256
13
14
15 szyfrant=TSzyfrant(89)
16 wiadomosc = int(input('Podaj dodatnią liczbę do zaszyfrowania: '))
17 szyfr=szyfrant.szyfruj(wiadomosc)
18 print('Wynik=%s' % szyfr)
19 print('Odszyfrowany: %s' % szyfrant.odszyfruj(szyfr))
```

Objaśnienie :

Utworzenie obiektu (wiersz szyfrant=TSzyfrant(89)) powoduje wywołanie konstruktora z parametrem 89. Takie nazwy rozpoczynające się i kończące od __ (dwóch podkreśleń) są w Pythonie stosowane do specjalnych identyfikatorów systemowych. Własność *klucz* nie musi być zapisana jawnie w kodzie - bo tworzy się ona dynamicznie - z chwilą wstawienia wartości (tu: w konstruktorze).

Moduły

Definicja klasy powinna być taka, aby jej użycie było jak najprostsze. Generujemy obiekt i wywołujemy odpowiednią metodę. To co jest w środku obiektu nie musi nas interesować. Skoro

tak – to najlepiej umieścić te szczegóły gdzie indziej – w innym pliku. Tak tworzy się moduły. Moduł w Pythonie to plik tekstowy z nazwą o rozszerzeniu „.py”.

Na naszym przykładzie – możemy stworzyć plik *cipher.py* z zawartością:

Przykład 5.4: **cipher.py**

```
class TSzyfrant:

    def __init__(self,klucz):
        self.klucz=klucz

    def szyfruj(self, liczba):
        return (liczba + self.klucz) % 256

    def odszyfruj(self, liczba):
        return (256 + liczba - self.klucz) % 256
```

Aby wykorzystać moduł, importujemy z niego to, co potrzebujemy – instrukcją import:

Przykład 5.4 c.d.: **cipher**

```
#encoding: UTF-8
from cipher import TSzyfrant

szyfrant=TSzyfrant(89)
wiadomosc = int(input('Podaj dodatnią liczbę do zaszyfrowania: '))
szyfr=szyfrant.szyfruj(wiadomosc)
print('Wynik=%s' % szyfr)
print('Odszyfrowany: %s' % szyfrant.odszyfruj(szyfr))
```

Dziedziczenie

Siła programowania obiektowego polega nie tylko na ukrywaniu szczegółów. Ważna jest możliwość tworzenia obiektów pochodnych. Czyli obiektów dziedziczących wszystkie cechy obiektu wcześniej zdefiniowanego. Dziedziczenie zaznaczamy dodając nazwę klasy z której dziedziczymy w nawiasach. Zobaczmy to na przykładzie modułu szyfr:

***Przykład** 5.5:* **inherit**

```
#encoding: UTF-8

class TSzyfrant:

    def __init__(self,klucz):
        self.klucz=klucz

    def szyfruj(self, liczba):
        return (liczba + self.klucz) % 256

    def odszyfruj(self, liczba):
        return (256 + liczba - self.klucz) % 256

class TSolitaire(TSzyfrant):

    def s_szyfruj(self, napis):
        wynik=[]
        for znak in napis:
            wynik.append(self.szyfruj(ord(znak)))
        return wynik

    def s_odszyfruj(self, zakodowany):
        wynik=''
        for kod in zakodowany:
            wynik= wynik+chr(self.odszyfruj(kod))
        return wynik
```

Klasa TSolitaire dziedziczy z klasy TSzyfrant metody kodowania liczb i wykorzystuje je do kodowania łańcuchów znaków. A oto jak możemy to wykorzystać:

```
from szyfr import TSolitaire
szyfrant=TSolitaire(88)
wiadomosc = raw_input('Podaj napis: ')
szyfr=szyfrant.s_szyfruj(wiadomosc)
print('Wynik=%s' % szyfr)
print('Deszyfr=%s' % szyfrant.s_odszyfruj(szyfr))
```

Pakiety

Moduły można umieszczać w podkatalogach (folderach), grupując tematycznie. Uzyskujemy w ten sposób dalsze uporządkowanie. Podkatalog zawierający plik o nazwie __init__.py (może być

pusty) Python traktuje jak pakiet modułów. Import z modułu umieszczonego w pakiecie polega na poprzedzeniu nazwy moduły nazwą pakietu (podkatalogu) z kropką na końcu. Na przykład gdy umieścimy moduł szyfr w katalogu/pakiecie szyfrownia, to import będzie wyglądał następująco:

```
from szyfrownia.szyfr  import TSolitaire
```

Nic innego w programie nie ulegnie zmianie.

Pakiety mogą być zagnieżdżone. Powiedzmy, że chcesz stworzyć pakiet o nazwie **swiat** zawierającą pakiety **azja**, **afryka** itd., zaś w nich na przykład **indie** czy **madagaskar**.

Oto, jak powinna wyglądać twoja struktura katalogów:

```
- <jakiś katalog wymieniony w sys.path>
  - swiat/
    - __init__.py
    - azja/
        - __init__.py
        - indie/
            - __init__.py
            - foo.py
    - afryka/
        - __init__.py
        - madagaskar/
            - __init__.py
            - bar.py
```

Aby użyć funkcji lub danej z modułu zawartego w pakiecie, wykonujemy import podając całą ścieżkę rozdzieloną kropkami.

Załóżmy, że mamy pakiet1 z modułami:

- modul11 zawierający funkcję fun11a i fun11b,
- modul2 zawierający funkcje fun12
- Oraz pakiet2 z modułem fun2.

Przykład ich użycia:

```
from pakiet1.modul11 import fun11a, fun11b
from pakiet1.modul12 import fun12
from pakiet2.modul2 import fun2

fun11a()
fun11b()
fun12()
fun2()
```

Główną rolą pliku __init__.py jest poinformowanie Pythona, że katalog zawiera pakiet modułów. Jednak plik ten nie musi być pusty. Możemy w nim zaimportować udostępniane elementy pakietu i dokonać zainicjowania zmiennych.

Jeśli na przykład w pakiet1.__init__.py wpiszemy:

```
from modul11 import fun11a,fun11b
from modul12 import fun12
```

To nasz testowy program możemy zmodyfikować następująco:

```
from pakiet1 import fun11a, fun11b, fun12
from pakiet2.modul2 import fun2

fun11a()
fun11b()
fun12()
fun2()
```

Dekoratory funkcji

Ponieważ funkcja jest w Pythonie obiektem, można dynamicznie zmieniać jej strukturę. Zostało to wykorzystane w mechanizmie zwanym dekoracją funkcji. Jest to stosunkowo nowy i dość zaawansowany sposób radzenia sobie ze złożonością. Dlatego w tym elementarnym podręczniku jedynie wspominamy o nim, podając jako przykład wykorzystania framework Flask (http://flask.pocoo.org/[27]). Prostą stronę internetową można w nim zaimplementować w kilku linijkach:

[27] http://flask.pocoo.org/

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"
if __name__ == "__main__":
    app.run()
```

Poprzedzenie definicji funkcji zapisem @app.route(”/”) mówi, że w rzeczywistości zostanie wykonana metoda route z obiektu app, a “udekorowana” funkcja hello dostarczy jedynie części treści do wygenerowania strony.

[v3]

Operacje na listach i łańcuchach znaków

Listy to podstawowa struktura danych używana przy przetwarzaniu danych w Pythonie. Jeśli mamy do czynienia z danymi pobieranymi ze źródeł zewnętrznych (na przykład odczytywanych z pliku CSV), to najczęściej pojawiają się one w postaci krotek. Natomiast dane lokalne - wytwarzane i zmieniane w programie są albo danymi elementarnymi (liczby, wartości logiczne), łańcuchami znaków albo listami.

Sekwencje

Listy, krotki i łańcuchy znaków mają wiele cech wspólnych. Są one **sekwencjami** – czyli zawierają ciąg uszeregowanych elementów. Na sekwencjach można wykonywać operacje:

- *testowanie członkostwa*, czyli in a not in wyrażeniach;
- *indeksowania*, czyli odniesienie do elementu sekwencji poprzez wskazanie numeru; na przykład lista[5] wskazuje szósty element listy (liczymy od zera);
- przeglądania przy pomocy instrukcji *for* (for element in lista).

Dla tych trzech rodzajów sekwencji wymienione powyżej (listy, krotki i łańcuchy), dostępna jest dodatkowo operacja *krojenia **(slice)***__, która pozwala nam uzyskać fragment sekwencji.

Sekwencję oznaczamy podając jej pierwszy element i pierwszy nie należący do sekwencji. Czyli **sekwencja[1:3]** zwraca fragment rozpoczynający się od elementu o numerze 1 (nie jest to pierwszy element, bo ten ma numer 0) a skończywszy na elemencie numer 2 (bo 3 oznacza pierwszy, który następuje **po** sekwencji.

Pominięcie zarówno dolnego jak i górnego ograniczenia ([:]) powoduje utworzenie kopii całej sekwencji.

Liczba ujemna oznacza końcowy element. O ile jednak sekwencja[1] oznacza drugi element (liczymy od zera), to sekwencja[-1] oznacza ostatni element.

Zobaczmy to na przykładach.

Przykład:

Przykład 6.1: **slices**

```
print([0,1,2,3,4])
print([0,1,2,3,4][1])
print([0,1,2,3,4][-1])
print([0,1,2,3,4][1:3])
print([0,1,2,3,4][:3])
print([0,1,2,3,4][1:3])
print([0,1,2,3,4][-1])
print([0,1,2,3,4][1:-1])
```

Wyjście:

```
[0, 1, 2, 3, 4]
1
4
[1, 2]
[0, 1, 2]
[1, 2]
4
[1, 2, 3]
```

Objaśnienie

Należy zwrócić uwagę na to, że pominięcie ograniczenia górnego lub dolnego sekwencji (na przykład [:3] nie ma dolnego ograniczenia) powoduje wzięcie sekwencji od początku lub do końca.

Przykład 6.2: **seq**

```
shoplist = ['jabłko', 'mango', 'marchew', 'banan']
name = 'Galicea'
#
print('lista zawiera chleb?', 'chleb' in shoplist)
print('lista zawiera banan?', 'banan' in shoplist)
# indeksowanie #
print('Element 0 to % s' % shoplist[0])
print('Element 1 to % s' % shoplist[1])
print('Element 2 to % s' % shoplist[2])
print('Element 3 to % s' % shoplist[3])
print('Element -1 to % s' % shoplist[-1])
print('Element -2 to % s' % shoplist[-2])
print('Znak 0 to % s' % name[0])
# fragmenty listy #
```

```
print('Elementy od 1 do 3 to %s' %  shoplist[1:3])
print('Elementy od 2 do końca to %s' % shoplist[2:])
print('Elementy od 1 do -1 to %s' % shoplist[1:-1])
print('Elementy od początku do końca to %s' % shoplist[:])
# fragmenty łańcucha #
print('znaki od 1 do 3 to %s' % name[1:3])
print('znaki od 2 do końca to %s' % name[2:])
print('znaki od 1 do -1 to %s' % name[1:-1])
print('znaki od początku do końca to %s' % name[:])
```

Wyjście:

```
('lista zawiera chleb?', False)
('lista zawiera banan?', True)
Element 0 to jabłko
Element 1 to mango
Element 2 to marchew
Element 3 to banan
Element -1 to banan
Element -2 to marchew
Znak 0 to G
Elementy od 1 do 3 to ['mango', 'marchew']
Elementy od 2 do końca to ['marchew', 'banan']
Elementy od 1 do -1 to ['mango', 'marchew']
Elementy od początku do końca to ['jab\xc5\x82ko', 'mango', 'marchew', 'banan']
znaki od 1 do 3 to al
znaki od 2 do końca to licea
znaki od 1 do -1 to alice
znaki od początku do końca to Galicea
```

Objaśnienie

Python numeruje sekwencje od 0 – dlatego shoplist[0] to pierwsza, a shoplist[3] – czwarta pozycja w shoplist.

Idenks może być też wartość ujemną, w tym przypadku pozycja jest obliczana od końca sekwencji. Dlatego shoplist[-1] odnosi się do ostatniego elementu w kolejności oraz shoplist[-2] pobiera przedostatni element w sekwencji.

Fragment shoplist[1:3] zwraca się fragment o sekwencji rozpoczynającej się od indeksu 1 (czyli drugiego elementu), a kończący na elemencie o indeksie 2, ale nie zawiera elementu o indeksie 3, który stanowi górne ograniczenie. Natomiast shoplist[:]zwraca kopię całej sekwencji.

Na przykład shoplist[:-1] powróci fragment sekwencji, która wyklucza ostatni element, ale zawiera całą resztę.

Referencje

Do czego może nam służyć kopia całej sekwencji ([:])? W języku Python działanie przypisania zmienna1=zmienna2 zależy od tego, co zawiera zmienna2. Jeśli są to dane proste (liczba, wartość logiczna), to w po tej operacji zmienna1 będzie zawierać kopię tych danych. Jeśli natomiast zmienna2 zawiera typ złożony, to wykonanie podstawienia powoduje, że zmienna *odnosi się* do tego samego obiektu! Oznacza to, że po dokonaniu podstawienia zmienne wskazują ten sam fragment pamięci komputera. Zawierają referencje do tego samego miejsca.

Przykład 6.3: **references**

```
shoplist = ['jabłko', 'mango', 'marchew', 'banan']
# mylist to tylko inna nazwa (referencja) tych samych danych!
mylist = shoplist
# usunięcie pierwszego elementu z shoplist kasuje go także dla mylist
del shoplist[0]
print('Usuwanie z referencji:')
print('lista wskazywana przez shoplist:', shoplist)
print('lista wskazywana przez mylist:', mylist)
# to samo po zrobieniu kopii
mylist = shoplist[:]
del mylist[0]
print('Usuwanie z kopii:')
print('lista wskazywana przez shoplist:', shoplist)
print('lista wskazywana przez mylist:', mylist)
```

Wyjście:

```
Usuwanie z referencji:
('lista wskazywana przez shoplist:', ['mango', 'marchew', 'banan'])
('lista wskazywana przez mylist:', ['mango', 'marchew', 'banan'])
Usuwanie z kopii:
('lista wskazywana przez shoplist:', ['mango', 'marchew', 'banan'])
('lista wskazywana przez mylist:', ['marchew', 'banan'])
```

Objaśnienie

Gdy usuwamy z referencji – zmiana jest dokonywana na wszystkich zmiennych wskazujących na to samo. Gdy usuwamy z kopii – oryginał nie jest zmieniany.

Łańcuchy znaków

A jak jest w przypadku łańcuchów znaków (napisów)? Gdy wpiszemy

```
a='abc'
b=a
```

to tworzymy kopię, czy przypisujemy do b tą samą referencję? Z punktu widzenia programisty to nie ma znaczenia, gdyż napisy nie mogą być modyfikowane (są „immutable"). Jak sobie zatem radzić z ich przetwarzaniem? Służą do tego funkcje, które przetwarzają łańcuch znaków i na wynik zwracają wynik tego przetworzenia. Łańcuch jest obiektem, a funkcje na nim działające są zdefiniowane dla[28] klasy [29] str[30]. Pełną listę takich funkcji (metod obiektu) zwraca polecenie help(str). Najbardziej użyteczne pokazuje poniższy przykład:

***Przykład** 6.4:* **strings**

```
# This is a string object
name = 'galicea'
#zamiana małych liter na wielkie
name = name.capitalize() #zamiana pierwszej litery na wielką
if name.startswith('Gal'):
    print('Tak - nazwa zaczyna się od "Gal"')

if 'a' in name:
    print('Tak - nazwa zawiera "a"')

if name.find('lic') != -1:
    print('Tak - nazwa zawiera łańcuch "lic"')

print(name.upper()) # duże litery
print(name.lower()) #zamiana wielkich liter na małe
print('Nazwa zawiera %s litery a' % name.count('a'))

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

***Wyjście**:*

[28] https://pl.python.org/docs/lib/string-methods.html
[29] https://pl.python.org/docs/lib/string-methods.html
[30] https://pl.python.org/docs/lib/string-methods.html

```
Tak - nazwa zaczyna się od "Gal"
Tak - nazwa zawiera "a"
Tak - nazwa zawiera łańcuch "lic"
GALICEA
galicea
Nazwa zawiera 2 litery a
Brazil_*_Russia_*_India_*_Chin
```

Objaśnienie

Przykład zawiera najważniejsze operacje na łańcuchach: sprawdzanie zawartości (find, startswith, in) i łączenia (join). Funkcja find zwraca -1 jeżeli tekst nie zawiera szukanego fragmentu. W przeciwnym przypadku zwraca indeks początku.

Zwykłe łączenie napisów można wykonać operatorem dodawania: ”abc”+”aaa” to tyle co ”abcaaa”. Nie można natomiast dodawać do łańcuchów innych typów (nie następuje automatyczna konwersja na napis). Zapis ”abc”+3 jest nielegalny. Ciekawą właściwością języka jest natomiast zapis ”abc”*3, który jest poprawny i oznacza powielenie tekstu (daje wynik ”abcabcabc”).

Użytecznym sposobem przetwarzania tekstów może niekiedy być zamiana ich na listy i odwrotnie:

```
# zamiana napisu na listę liter:
list('aaa')
# zamiana napisu na listę słów
'aaa bbb ccc'.split(' ')
# połączenie listy w napis:
''.join(['a','b','c'])
```

Czasem potrzebna jest też konwersja znaków na kody (cyfry za pomocą których jest kodowany napis w pamięci komputera) i odwrotnie.

```
#zamiana znaku na kod ascii
print(ord('a'))
#zamiana kodu ascii na znak
print(chr(97))
```

Python akceptuje łańcuchy znaków zapisane w wielu liniach. W takim przypadku należy apostrof (’ lub ”) początkowy i końcowy powtórzyć trzykrotnie:

```
"""napis
w dwóch liniach"""
```

Jeśli w łańcuchu występuje apostrof ograniczający napis – należy poprzedzić go znakiem . Czyli ”apostrof to znak \” ” oznacza to samo co ’apostrof to znak ” ’.

Formatowanie napisów

W niniejszym podręczniku często występuje składanie (formatowanie) łańcucha znaków z wzorca i danych. Miejsca we wzorcu, w których mają zostać wstawione dane zaznaczmy znakami %s:

```
"tekst zawierający %s oraz %s – w domyślnym formacie" %  (1,3)
```

To jednak nie są jedyne możliwości.

Jeśli chcemy aby liczby miały odpowiednią ilosć cyfr i miejsce dziesiętne, możemy w miejsce %s użyć %f:

```
>>> "tekst zawierający %.1f oraz %5.2f – w zdefiniowanym formacie" %  (1,3)
'tekst zawierający 1.0 oraz  3.00 – w zdefiniowanym formacie'
```

Więcej możliwości: https://pl.python.org/docs/lib/typesseq-strings.html[31]

W powyższych przypadkach kolejność użycia zmiennych do wypełnienia wzorca wynika z ich kolejności w krotce (sekwencji). Możemy jednak użyć słownika, wskazując miejsce w formacie nazwami kluczy ujętymi w nawiasy okrągłe:

```
>>> "tekst zawierający %.1f oraz %5.2f – w zdefiniowanym formacie" %  (1,3)
'tekst zawierający 1.0 oraz  3.00 – w zdefiniowanym formacie'
>>> "tekst zawierający %(liczba2).1f oraz %(liczba1)5.2f – w zdefiniowanym forma\
cie" %  {'liczba1':1, 'liczba2': 3}
'tekst zawierający 3.0 oraz  1.00 – w zdefiniowanym formacie'
```

[v3]

[31] https://pl.python.org/docs/lib/typesseq-strings.html

Definiowanie i używanie funkcji

Poprzednie rozdziały zawierają elementarną wiedzę na temat programowania w języku Python. Jest ona wystarczająca do rozpoczęcia samodzielnego tworzenia programów. Każdy może samemu testować tą wiedzę na prostych przykładach – takich jak zawarto w treści podręcznika. W przykładach tych pominięto niektóre mniej istotne zagadnienia. Szczególnie dotyczy to podstawowego budulca programów, jakim są funkcje. Obecnie przyjrzymy się im bliżej.

Wprowadzenie

Samo określenie „**funkcja**" pochodzi z języka matematyki. Na przykład zapis **sin(x)** oznacza funkcję sinus z parametrem x. Możemy przyjąć, że za nazwą **sin** kryje się obliczenie wartości funkcji dla określonego parametru. Ta idea została w informatyce zaakceptowana i poszerzona. Funkcją nazywa się oznaczony nazwą fragment kodu, zwracający jakieś wyniki dla określonych parametrów.

Funkcje definiuje się używając słowa kluczowego **def**. Po nim następuje nazwa identyfikująca funkcję, następnie para nawiasów, które mogą zawierać kilka nazw zmiennych, a na końcu dwukropek. Poniżej zaczyna się blok instrukcji, które są częścią tej funkcji wykonywaną po wywołaniu (użyciu) funkcji.

***Przykład** 7.1:* **f_simple**

```
1  # -*- coding: utf-8 -*-
2
3  def powiedzAhoj():
4      print 'Ahoj, przygodo!' # Blok należący do funkcji.
5  # Koniec funkcji.
6
7  powiedzAhoj() # Wywołanie funkcji.
8  powiedzAhoj() # Ponowne wywołanie funkcji.
```

Rezultat:

```
Ahoj, przygodo!
Ahoj, przygodo!
```

Objaśnienie:

Definiujemy funkcję *powiedzAhoj* używając składni opisanej wcześniej. Ta funkcja nie wymaga żadnych parametrów i w związku z tym nie ma żadnych nazw (zmiennych) zadeklarowanych w nawiasach. Funkcję możemy wywołać wielokrotnie (tu: dwukrotnie) – w różnych miejscach. Może to pomóc w unikaniu powielania tego samego lub podobnego (parametryzowalnego) kodu wielokrotnie.

Parametry funkcji

Funkcja może posiadać parametry, czyli wartości, które dostarcza się do niej. Parametry są podobne do zmiennych, różnią się jedynie tym, że pierwsze przypisywanie wartości (zainicjowanie) następuje podczas wywoływania funkcji. Inicjowanie to może następować poprzez podanie wartości (wyrażenia) lub zmiennej. Jeśli do parametru jest przekazywane wyrażenie (także łańcuch znaków), to zmiana parametru wewnątrz funkcji oczywiście nie rodzi żadnych skutków na zewnątrz. Jeśli przypomnimy sobie problem referencji z poprzedniego rozdziału, to możemy się domyślić co będzie, gdy jako parametr przekażemy zmienną zawierającą listę danych. Oczywiście zostanie utworzony parametr funkcjonujący tak jak zmienna, ale wskazujący na ten sam obszar pamięci, jak zmienna przekazywana do funkcji. Poniższy przykład powinien to wyjaśnić:

***Przykład** 7.2:* **f_parameters**

```
1  def suma1(a,b):
2      print(a+b)
3      a=a+b
4
5  def suma2(a):
6      print(a[0]+a[1])
7      a[0]=a[0]+a[1]
8
9  suma1(3,2*3)
10 a=3
11 suma1(a,6)
12 print('parametr prosty po wyjsciu=',a)
13 suma2([3,6])
14 lista=[3,6]
15 suma2(lista)
16 print('parametr / lista po wyjsciu=',lista)
```

Wynik:

```
9
9
('parametr prosty po wyjsciu=', 3)
9
9
('parametr / lista po wyjsciu=', [9, 6])
```

Objaśnienie

1. Należy zwrócić uwagę na ostatnią linię wyniku. Pokazuje ona, że po wywołaniu funkcji zmianie uległa zawartość zmiennej lista. Inaczej niż w przypadku zmiennej a, która zawiera dane prostego typu.
2. Zauważmy, że każda z powyższych funkcji została wykonana dwukrotnie. To pokazuje jak trafne jest użyte określenie „parametr". Blok kodu wewnątrz funkcji jest wykonywany dla różnych wartości – czyli zostaje „sparametryzowany".

Parametry i argumenty

Często stosuje się konwencję nazewnictwa, ograniczającą użycie określenia „**parametr**" do definicji funkcji. Wartość nadawana parametrom w chwili wywołania nazywa się „**argumentami**". Zamiast więc mówić funkcja „*sumuj1*" została wywołana z parametrami 3 i 6 (co nie jest błędem, choć kogoś może razić) – mówimy, że funkcja ta została wywołana z argumentami 3 i 6.

Zobaczmy jeszcze jeden przykład, w którym wykorzystanie funkcji **if** sprawia, że dla różnych argumentów sterowanie wewnątrz funkcji przebiega nieco inaczej:

***Przykład** 7.3:* f_arguments

```
1  def wypiszMax(a, b):
2      if a > b:
3          print('%s to maksimum' % a)
4      elif a == b:
5          print('liczby jednakowe')
6      else:
7          print('maksimum to %s' % b)
8
9  wypiszMax(3, 4)
10 wypiszMax(7, 5)
```

Wynik:

```
maksimum to 4
7 to maksimum
```

Objaśnienie:

Definiujemy funkcję wypiszMax, w której wypisujemy na ekran największą z liczb a i b za pomocą prostej kombinacji **if**...**elif**...**else**.

Podczas pierwszego uruchomienia wypiszMax dostarczamy bezpośrednio wartości do parametrów (czyli argumenty). Za drugim razem wywołujemy ją używając dwóch zmiennych. Linijka wypisz-Max sprawia, że wartość argumentu x zostaje przekazana jako parametr a, a wartość argumentu y jako parametr b.

Wynik funkcji - wyrażenie return

Wyrażenia return używamy do wyjścia z funkcji. Możemy opcjonalnie zwrócić w tym momencie jakąś wartość.

***Przykład** 7.4:* **f_return**

```
1 def maximum(x, y):
2     if x > y:
3         return x
4     else:
5         return y
6 
7 print maximum(2, 3)
```

Rezultat:

```
1 3
```

Objaśnienie:

Funkcja maximum wyszukuje maksymalną wartość spośród podanych. Używa w tym celu prostej konstrukcji if...else a następnie zwraca tę wartość dzięki poleceniu return. Zauważ, że użycie wyrażenia return bez wartości jest równoznaczne z użyciem return None. None to specjalny typ w Pythonie, który reprezentuje po prostu nic. Używa się tego na przykład, gdy się chce przekazać, że zmienna nie ma wartości. Każda funkcja domyślnie otrzymuje na końcu return None, chyba że napiszesz własne return. Możesz to sprawdzić chociażby w ten sposób:

```
def funkcja():
    pass
print funkcja()
```

Wyrażenie pass wskazuje na pusty blok wyrażeń.

Domyślne wartości parametrów

W Pythonie istnieje możliwość zdefiniowania części parametrów jako opcjonalnych. Wówczas musimy podać wartość domyślną, którą parametr przyjmuje, jeśli nie został podany odpowiadający mu argument przy wywołaniu funkcji. Domyślną wartość parametru definiuje się tak jak przypisywanie wartości zmiennej – z użyciem znaku = (przypisania) po którym następuje wyrażenie, jakiego wartość ma zostać przypisana parametrowi, gdy nie zdefiniowano tej wartości (argumentu) przy wywołaniu funkcji. Choć Python dopuszcza definiowania domyślnej wartości jako zmiennego wyrażenia (zależnego o zmiennych i funkcji), to nie należy z tej możliwości korzystać. Wartość domyślna parametru powinna być niezmienna.

```
def powiedz(wiadomosc, ile = 1):
    print wiadomosc * ile

powiedz('Ahoj')
powiedz('Przygodo!', 5)
```

Rezultat:

```
Ahoj
Przygodo!Przygodo!Przygodo!Przygodo!Przygodo!
```

Objaśnienie:

Funkcja powiedz ma wypisywać podaną jej wiadomość określoną ilość razy. Jeżeli nie podamy wartości, napis zostanie wyświetlony domyślnie, czyli raz. Osiągnęliśmy to przez przypisanie domyślnej wartości 1 do parametru ile.

Przy pierwszym użyciu tej funkcji podaliśmy jedynie napis, który dzięki temu został wyświetlony jedynie raz. Za drugim razem, oprócz napisu dołożyliśmy argument 5, a zatem uzyskaliśmy 5 powtórzeń napisu.

Ważne!

Domyślne wartości mogą posiadać jedynie te parametry, które są na końcu listy. Innymi słowy, nie możesz nadać wartości domyślnej parametrowi, jeżeli po nim wystąpi taki, który nie będzie jej posiadał (idąc wzdłuż ciągu parametrów danej funkcji). Jest to spowodowane tym, że wartości są przypisywane według kolejności występowania parametrów. Na przykład możesz napisać def funkcja(a, b = 5), ale nie możesz napisać **def funkcja(a = 5,b)**!

Wywołanie funkcji z odwołaniem do parametrów poprzez nazwę

Przypisanie wartości argumentów poszczególnym parametrom następuje w kolejności od lewej do prawej. Jeżeli jednak w definicji funkcji podano domyślną wartość więcej niż jednego parametru – to możemy chcieć przekazać wartość tylko do jednego z nich i to niekoniecznie tego pierwszego z lewej.

Mówi się czasem, że są to argumenty ze słowem kluczowym - ale to określenie jest mylące, gdyż chodzi o nazwy definiowane przez programistę, a nie słowa kluczowe języka.

***Przykład** 7.6:* **f_optional**

```
1  def f(a, b = 5, c = 10):
2      print 'a wynosi', a, 'zaś b wynosi', b, 'a c wynosi', c
3
4  f(3, 7)
5  f(25, c = 24)
6  f(c = 50, a = 100)
```

Rezultat:

```
a wynosi 3 zaś b wynosi 7 a c wynosi 10
a wynosi 25 zaś b wynosi 5 a c wynosi 24
a wynosi 100 zaś b wynosi 5 a c wynosi 50
```

Objaśnienie:

Ta funkcja posiada jeden parametr bez domyślnej wartości i dwa, które ją posiadają. Za pierwszym razem wpisujemy f(3, 7), dzięki czemu parametr a dostaje wartość 3, parametr b dostaje 7, a parametr c domyślną wartość 10. Za drugim razem wpisujemy f(25, c = 24), więc parametr a otrzymuje wartość 25 (zgodnie z kolejnością), zaś parametr c dostaje 24, dzięki słowu kluczowemu. Parametr b ma domyślną wartość 5.

Za trzecim razem wpisujemy f(c = 50, a = 100) używając jedynie słów kluczowych. Zauważ, że przypisujemy wartość najpierw parametrowi c, niezależnie od tego, że parametr a występuje wcześniej w deklaracji funkcji.

Zmienna ilość parametrów

Funkcje z domyślną wartością parametrów mogą być wywoływane z różną ilością argumentów. Jednak lista argumentów nie być dowolna – bo musi się zgadzać z listą zdefiniowanych parametrów.

Istnieje jednak sposób definiowania funkcji o zupełnie dowolnej liście parametrów. Poprzedzenie nazwy parametru gwiazdką (*) oznacza dowolną ilość nie nazwanych argumentów, widocznych w postaci krotki. Dwie gwiazdki (**) oznaczają słownik z argumentami przekazanymi wraz z nazwą parametrów.

***Przykład** 7.7:* **f_variant**

```
1 def f(*par,**kpar):
2     print par
3     print kpar
4 
5 f(1,2,4)
6 f('1',a=2,b=3)
```

Wyjście:

```
(1, 2, 4)
{}
('1',)
{'a': 2, 'b': 3}
```

Objaśnienie:

W pierwszym wywołaniu uzyskaliśmy krotkę trzech wartości. W drugim mamy argumenty z nazwą – więc zostały one umieszczone w strukturze słownika. Tylko pierwszy argument jest przekazany w formie krotki.

Zakres widoczności parametrów i zmiennych

Częstą przyczyną błędów w programach jest użycie zmiennej poza zakresem jej widoczności. W obrębie funkcji każda użyta zmienna jest z reguły lokalna – czyli nie jest ma nic wspólnego z ewentualnymi zmiennymi o takich samych nazwach, ale użytymi poza blokiem (ciałem) funkcji.

W funkcji można używać nazw zmiennych na kilka sposobów:

1. Najczęściej są to nazwy parametrów.
2. Poza parametrami można w bloku funkcji tworzyć nowe zmienne. Są one widoczne tylko w obrębie funkcji. Po zakończeniu wykonania funkcji przestają one istnieć. Tym bardziej nie ma znaczenia sama definicja funkcji.
3. Istnieje jednak sposób na to, by wewnątrz funkcji uzyskać dostęp do zmiennych zdefiniowanych w głównym bloku funkcji. Służy temu opisane poniżej słowo kluczowe global.

4. Lepszym (zdecydowanie) od użycia global sposobem dostępu do zmiennych spoza funkcji jest przekazywanie poprzez parametry struktur (na przykład list) i obiektów. Jeśli dokonamy zmian we wskazanym przez parametr obiekcie, to dokonane zmiany pozostaną po zakończeniu funkcji. Nazwy zmiennych i metod ze środka obiektu poprzedzamy nazwą tego obiektu (lub parametru wskazującego na ten obiekt) i kropką. Temu zagadnieniu zostanie poświęcony kolejny rozdział podręcznika.
5. Identyfikatory pochodzące z innych modułów pojawiają się wskutek wykonania instrukcji import. Są one widoczne w całym module od chwili wykonania instrukcji import. A więc także wewnątrz funkcji zdefiniowanych w tym module!!

Zmienne lokalne

Oczywiście nazwy parametrów są widoczne tylko w obrębie funkcji. Użycie takiej samej nazwy poza funkcją nie prowadzi do konfliktów. To nie są te same nazwy / zmienne, choć są takie same. Pokazuje to poniższy prosty przykład. Jest rzeczą niezmiernie ważną, aby go dobrze zrozumieć.

Przykład 7.8: f_local

```
1  x = 50
2
3  def f(x):
4      print 'x wynosi', x
5      x = 2
6      print 'Zmieniono lokalne x na', x
7
8  f(x)
9  print 'x wynosi nadal', x
```

Rezultat:

```
x wynosi 50
Zmieniono lokalne x na 2
x wynosi nadal 50
```

Objaśnienie:

Parametr x ona widoczny tylko wewnątrz funkcji. Na zewnątrz mamy inną zmienną o takiej samej nazwie. Z chwilą wywołania funkcji, wartość zmiennej x zostaje przepisana do parametru x. Zmiana tego parametru (nadanie mu wartości 2) nie powoduje żadnych skutków dla zawartości zmiennej x, której nadaliśmy wartość 50. Końcowe wywołanie print, pokazuje wartość x w bloku głównym i dowodzi, że nie została ona zmieniona.

Tak samo jak parametry są traktowane zmienne lokalne. Jeśli zmienimy nazwę parametru na inną (oczywiście musi to dotyczyć także pierwszego print w funkcji) – działanie programu nie zmieni się!

Przestrzenie nazw

W powyższym akapicie mówi się raz o zmiennych, a innym razem o nazwach zmiennych. To jak to w końcu jest? Chodzi o zakres widoczności identyfikatorów, czy zmiennych na jakie wskazują? Przyjęło się mówić o zakresach widoczności zmiennych lub parametrów. Każda zmienna ma swój zakres widoczności, czyli blok funkcji albo obiekt w została ta zmienna zdefiniowana (w Pythonie – przez pierwsze jej użycie / zainicjowanie). Ta tradycja nie jest jednak najlepsza, gdyż chyba bardziej zrozumiałe (a przede wszystkim bardziej uniwersalne) jest operowanie pojęciem przestrzeni nazw (miejsce „zakresu widoczności").

Pojęcie **przestrzeni nazw** (ang. **namespace**) nie pochodzi od twórców języka Python, ale jest bardziej uniwersalne. To bardziej precyzyjne określenie kontekstu. Analogicznie jak w języku potocznym - to samo słowo może oznaczać co innego w różnych kontekstach. Zakłada się jednak, że w danej przestrzeni nazw użyte identyfikatory są niepowtarzalne. Natomiast w różnych przestrzeniach nazw mogą istnieć takie same identyfikatory, ale oznaczające niekoniecznie to samo.

Możemy zatem mówić o przestrzeni nazw klasy i obiektu a także funkcji. To tylko wygodne pojęcie określające jaki jest zasięg widoczności identyfikatorów. Nie kryją się za nim żadne specjalne mechanizmy języka Python.

Odwołanie do identyfikatorów z innej przestrzeni nazw jest czasem możliwe poprzez dodanie przedrostka z identyfikatorem tej przestrzeni – na przykład nazwy klasy lub obiektu (temu będzie poświęcony następny rozdział).

Używanie pojęcia przestrzeni nazw pozwala lepiej zrozumieć to, że należy odróżnić znaczenie identyfikatorów (dostępność i znaczenie w programie) od tego co oznaczają (zmienne, obiekty etc). Przestrzeń nazw w jakiej używamy identyfikatorów decyduje wyłącznie o tym, czy są one dostępne i sensowne. Natomiast dostęp do pamięci komputera z użyciem tych identyfikatorów to trochę inne i bardziej złożone zagadnienie. Programistę siłą rzeczy bardziej interesuje to, co przy użyciu nazwy może wykonać (na przykład zainicjować oznaczoną tą nazwą zmienną). Ale na przykład fakt, że po wyjściu z programu znikają zmienne lokalne nie wynika z tego, że opuszczamy przestrzeń nazw, ale z zasad zarządzania pamięcią w Pythonie. Jeśli zmienna wskazuje na strukturę (lista, obiekt), to jej zniknięcie nie musi się wiązać ze zniknięciem struktury. Struktura znika dopiero gdy znikną wszystkie zmienne na nią wskazujące! Widać więc, że nazwy żyją sobie (w swoich przestrzeniach nazw) a zmienne sobie (w swoich zakresach widoczności / istnienia) ;-).

Powyższa dygresja w zasadzie nie ma znaczenia dla zrozumienia mechanizmów Pythona. Jednak każdy aktywny programista na pewno spotka się nie raz z tym zagadnieniem – więc warto przy okazji je sobie przyswoić.

Użycie wyrażenia global

Jeżeli wewnątrz funkcji chcemy użyć zmiennej zdefiniowanej w głównym bloku programu (czyli poza funkcjami i klasami), możemy wskazać Pythonowi, że nazwa nie jest lokalna,ale globalna za pomocą słowa kluczowego global.

Załóżmy dla przykładu, że blok funkcji rozpoczyna się od instrukcji:

```
global x
x=1
```

Gdyby nie było instrukcji ze słowem global – podstawienie do x jedynki spowodowałoby zainicjowanie zmiennej lokalnej (lub zmianę parametru x – jeśli taki zdefiniowano). Użycie global sprawia, że dokonujemy zmiany zmiennej globalnej.

Jednakże to nie jest najlepszym rozwiązaniem, ponieważ wtedy czytający program musi dla jego zrozumienia szukać zmiennej poza blokiem funkcji. Użycia wyrażenia ze słowem kluczowym global należy zatem unikać. Wiedza z tego podrozdziału powinna być więc przydatna ewentualnie do analizy cudzych, najczęściej niezbyt dobrze napisanych programów ;-).

***Przykład** 7.9:* **f_global**

```
1  x = 50
2
3  def f():
4      global x
5      print 'x wynosi', x
6      x = 2
7      print 'Zmieniono globalne x na', x
8
9  f()
10 print 'Wartość x wynosi', x
```

Rezultat:

```
x wynosi 50
Zmieniono globalne x na 2
Wartość x wynosi 2
```

Objaśnienie:

Używamy wyrażenia global w celu zadeklarowania, że x jest zmienną globalną — w związku z tym, gdy przypisujemy jakąś wartość do x we wnętrzu funkcji, widać tę zmianę również wtedy, gdy używamy x w głównym bloku programu. Możesz określać więcej niż jedną zmienną globalną w tej samej linijce, na przykład global x, y, z.

Świat obiektów

W rozdziale dotyczącym radzenia sobie ze złożonością, przedstawiono obiekty jako jeden ze sposobów budowania programu z mniejszych części. Wyróżniliśmy definicję obiektu – czyli **klasę**, od samego obiektu – czyli „**instancji**" klasy (obiekt utworzony zgodnie z definicją). Korzystając ze stworzonej (samemu lub przez społeczność programistów) definicji klasy, można wykreować wiele obiektów – stosowanie do potrzeb. Obiekty te różnią się od siebie wartością własności (czyli zmiennych zawartych w obiekcie) i sposobem użycia. Zmienne wewnątrz obiektu nazywamy polami lub własnościami. Użycie obiektu polega na operowaniu na własnościach (odczyt, zmiana) oraz wywoływaniu funkcji zdefiniowanymi w klasie obiektu. Funkcje te – zwane metodami klasy - definiują możliwe zachowania obiektów danej klasy. Stosowanie obiektów pozwala na ukrycie szczegółów implementacji. Możemy traktować je jak „czarną skrzynkę". Ich stosowanie nie wymaga wiedzy na temat tego co jest w środku. Nie wnikamy w to, jak funkcje (metody) obiektu zostały zaimplementowane. Ważne jest tylko to, że wywołanie takiej metody w określonym stanie daje określony efekt.

Analogicznie postrzegamy świat realny. Włączamy telewizor i nie zastanawiamy się w tym momencie jak to działa. Telewizor to taki obiekt z określonymi funkcjami dostępnymi przy użyciu pilota. Postrzeganie świata przez pryzmat obiektów okazuje się bardzo uniwersalne. Dlatego programowanie obiektowe jest intuicyjnie proste. Przynajmniej jeśli chodzi o użycie gotowych klas.

W tym rozdziale zajmiemy się jednak także nieco trudniejszą kwestią jaką jest definiowanie klas. Na początek dwa przykłady ilustrujące różnicę między klasą a obiektem. Pierwszy należałoby opatrzyć komentarzem „nie róbcie tego w domu". Tak się po prostu nie powinno programować – tu robimy to wyłącznie dla odsłonięcia pewnych cech klas i obiektów.

Przykład 8.1: **o_sumator**

```
1  class Sumator():
2      wynik = 0
3      krok=1
4
5      def dodaj(self, ile):
6          self.wynik+=(ile*self.krok)
7          return self.wynik
8
9  Sumator.wynik=1
10 sumator1=Sumator()
11 Sumator.wynik=2
12 Sumator.krok=2
13 sumator2=Sumator()
```

```
14
15 for ilosc_krokow in (1,1,2):
16     print('sumator1: %s' % sumator1.dodaj(ilosc_krokow))
17     print('sumator2: %s' % sumator2.dodaj(ilosc_krokow))
18
19 print('wlasnosc klasy: %s' % Sumator.wynik)
20 print('metoda klasy dla obiektu sumator1: %s' % Sumator.dodaj(sumator1,1))
```

Wyście:

```
sumator1: 4
sumator2: 4
sumator1: 6
sumator2: 6
sumator1: 10
sumator2: 10
wlasnosc klasy: 2
metoda klasy dla obiektu sumator1: 12
```

Objaśnienie:

Choć przykład nie jest zbyt złożony, to nawet doświadczonym programistom może sprawić problem przewidzenie wyników na podstawie kodu programu. Klasa Sumator można służyć do pamiętania (we własności o nazwie wartosc) odległości pokonanej krokami długości „krok" (inna własność obiektu).

To czego nie powinno się robić – to stosowanie zmian w definicji klasy poza tą klasą. Gdzie potem w dużym kodzie szukać tych zmian? Nie powinniśmy więc tworzyć takich instrukcji:

```
Sumator.wynik=1
Sumator.wynik=2
Sumator.krok=2
```

Na dodatek widzimy, że pierwsza z powyższych linii okazała się w programie bez znaczenia. Jeśli ktoś dobrze pozna język Python to zrozumie dlaczego. Ale taka wiedza nie jest niezbędna dla rozpoczęcia zabawy w programowanie ;-). Tworząc obiekt nie wpisujemy do niego kopii definicji klasy. A więc wstawienie do własności o nazwie „wynik" liczby 2 ma takie samo znaczenie dla obu obiektów (sumator1 i sumator2). Nie ma zaś znaczenia to, że przed wygenerowaniem pierwszego obiektu zmieniliśmy własność dla klasy (bo ta zmiana została „przykryta"). Równie ciekawie i równie źle wygląda ostatnia linijka programu. Metody klasy zawsze są wywoływane dla jakiegoś obiektu (instancji klasy). Jeśli nie stosujemy notacji obiekt.metoda(), to musimy podać w pierwszym parametrze ten obiekt. Tylko po co? Na szczęście nie jest możliwe wywołanie metody klasy bez podania instancji: **Sumator.dodaj(1)**. To tylko zaciemniłoby różnicę między obiektem i klasą.

Jeśli pomimo wyjaśnień powyższy przykład jest niezrozumiały, to nie ma się czym przejmować. Poniżej napiszemy go tak, jak się programować powinno. Obiekty mają przecież uprościć a nie zagmatwać kod programu.

Przykład 8.2: **o_sumator2**

```
1  class Sumator2():
2      wynik = 0
3      krok = 0
4
5      def __init__(self,start,krok):
6          self.wynik=start
7          self.krok=krok
8
9      def dodaj(self, ile):
10         self.wynik+=(ile*self.krok)
11         return self.wynik
12
13 sumator1=Sumator2(1,2)
14 sumator2=Sumator2(2,2)
15
16 for ilosc_krokow in (1,1,2):
17     print('sumator1: %s' % sumator1.dodaj(ilosc_krokow))
18     print('sumator2: %s' % sumator2.dodaj(ilosc_krokow))
```

Wyjście:

```
sumator1: 3
sumator2: 4
sumator1: 5
sumator2: 6
sumator1: 9
sumator2: 10
```

Objaśnienie:

Teraz wszystko powinno być intuicyjnie zrozumiałe. Zdefiniowaliśmy klasę Sumator2 i utworzyli dwa obiekty tej klasy, inicjując ich własności z chwilą tworzenia. Służy do tego specjalna funkcja zdefiniowana w klasie obiektu zwana konstruktorem (lub kreatorem). Nosi ona nazwę **__init__** i jest wywoływana za każdym razem, gdy tworzymy obiekt. Zapis Sumator(2,2) oznacza: utwórz obiekt i wywołaj konstruktora __init__ z parametrami (2,2).

Ponieważ własności są inicjowane w konstruktorze – dwie pierwsze linijki definicji klasy (wynik = 0 i krok = 0) są zbędne. Pozostawiamy często takie inicjowanie wyłącznie dla jasności kodu (aby nie musieć analizować kodu konstruktora, aby ustalić jakie obiekt danej klasy ma własności.

Poprzedzając nazwę zmiennych słowem self wskazujemy, że chodzi o własność obiektu, a nie o zwykłą zmienną lub parametr. Także definicja każdej funkcji będącej metodą klasy powinna zawierać jako pierwszy parametr self - jest przez niego przekazywany obiekt na rzecz którego wywołujemy funkcję.

Mamy więc dwa magiczne momenty (dziejące się poza naszym kodem) na które należy zwrócić uwagę:

- z chwilą tworzenia obiektu wywoływany jest konstruktor __init__
- z chwilą wywołania metody dla danego obiektu, zostaje on przekazany przez dodatkowy parametr self (nazwa stosowana zwyczajowo – ale nie spotyka się raczej jej zmiany).

W naszym przykładzie nie ma w ogóle odniesień bezpośrednich do klasy obiektu! Taki styl programowania jest bardzo polecany w Pythonie.

Adres zwrotny

Wróćmy na chwilę do magicznego słówka **„self”**. Tak nazywamy parametr, którym metody klas różnią się od zwykłych funkcji. On jest zawsze umieszczany na początku listy parametrów. Ten parametr odnosi się do samego obiektu. Dzieje się tak bez dodatkowych zapisów w programie. Python robi to samemu.

Załóżmy, że masz klasę **MojaKlasa** oraz instancję tej klasy o nazwie mojobiekt. Gdy wywołujesz metodę tego obiektu jako **mojobiekt.metoda(arg1, arg2)**, Python automatycznie tłumaczy to na **MojaKlasa.metoda(mojobiekt, arg1, arg2)**. Właśnie dlatego musimy dodawać parametr self – który w tym wypadku przyjmuje wartość mojobiekt.

Oznacza to także, że nawet jeśli napiszesz metodę, która będzie wywoływana bez argumentów, to i tak musisz ją zdefiniować jeden parametr – self – wskazujący na obiekt. Z uwagi na to, że self wskazuje on miejsce (obiekt) wywołania metody – nazywa się go czasem adresem zwrotnym.

Uporządkujmy naszą wiedzę o obiektach

Zaczęliśmy od trudnego przykładu – pora więc na coś łatwiejszego. Obiekty to bardzo istotna rzecz, więc nawet jeśli wydają się zrozumiałe – warto analizując proste przykłady podsumować naszą wiedzę w tej materii.

Klasy

Oto najprostsza możliwa klasa:

```
class Osoba:
    pass # Pusty blok

o = Osoba()
print o
```

Rezultat:

```
<__main__.Osoba instance at 0xb7cb542c>
```

Objaśnienie: Nową klasę tworzymy za pomocą polecenia class i nazwy klasy. Potem następuje wcięty blok poleceń, które składają się na ciało klasy. W naszym przypadku blok ten jest pusty, co sygnalizujemy poleceniem pass.

Następnie tworzymy obiekt tej klasy przez użycie nazwy klasy i pary okrągłych nawiasów. Dla sprawdzenia potwierdzamy typ zmiennej przez wypisanie jej na ekran. Python mówi nam, że mamy instancję klasy Osoba w module __main__ (tak nazywa się automatycznie moduł, który jest wykonywany jako pierwszy / główny).

Zauważmy, że jest wypisywany także adres w pamięci komputera, gdzie ten obiekt jest przechowywany. Oczywiście adres ten będzie inny przy każdym uruchomieniu programu, ponieważ Python może przechowywać obiekty, gdzie tylko znajdzie na nie miejsce.

Metody obiektowe

Jak już była mowa, klas/obiekty mogą posiadać metody, które są niczym innym, jak funkcjami z dodatkowym argumentem self. Teraz zobaczymy przykład takiej metody.

Przykład 8.3: **o_methods**

```
class Osoba:
    def przywitajSie(self):
        print 'Witaj, jak się masz?'

o = Osoba()
o.przywitajSie()
```

Rezultat:

```
Witaj, jak się masz?
```

Objaśnienie:

Ten krótki przykład można także zapisać jako **Osoba().przywitajSie()**

W tym przykładzie widzimy adres zwrotny self w akcji. Zauważmy, że metoda przywitajSie nie wymaga żadnych argumentów, a mimo to w jej definicji widnieje self. Metoda __init__ Istnieje wiele nazw metod, które mają specjalne znaczenie dla klas w Pythonie. Najczęściej takie wbudowane funkcje specjalne mają identyfikatory zaczynające się i kończące podwójnymi podkreślnikami. Metoda __init__, która jest wywoływana w momencie, kiedy tworzony jest obiekt danej klasy. Jest ona przydatna, kiedy chcemy zainicjować obiekt w jakiś sposób.

Przykład 8.4: **o___init__**

```
1 class Osoba:
2     def __init__(self, imie):
3         self.imie = imie
4     def przywitajSie(self):
5         print 'Witaj, mam na imię', self.imie
6 
7 o = Osoba('Swaroop')
8 o.przywitajSie()
```

Rezultat:

```
Witaj, mam na imię Swaroop
```

Objaśnienie:

Definiujemy metodę __init__ z parametrem imie (oprócz obowiązkowego **self**). Tworzymy w niej także pole o nazwie imie. Zauważmy, że są to dwie różne zmienne, mimo że mają tę samą nazwę. Użycie notacji z kropką umożliwia nam rozróżnienie ich od siebie.

Najważniejsze w tym przykładzie jest jednak to, że nie wywołujemy jawnie metody __init__, ale tylko podczas tworzenia nowego obiektu podajemy argumenty owej metody w nawiasach po nazwie klasy. Na tym właśnie polega specjalne znaczenie metody __init__.

Dzięki przeprowadzeniu inicjowania naszego obiektu, możemy używać pola self.imie w metodach tej klasy, jak zrobiono to w metodzie przywitajSie. Własności klas a przestrzenie nazw Pola (własności) obiektów, są zmiennymi mającymi sens jedynie w przestrzeni nazw danej klasy. Oznacza to, że możemy używać ich nazw tylko w kontekście danej klasy czy obiektu.

Przykład 8.5: **o_robot**

```
class Robot:
  u'''Reprezentuje robota, z nazwą.'''

  def __init__(self, nazwa):
    u'''Inicjalizuje dane.'''
    self.nazwa = nazwa
    print '(Inicjalizacja %s)' % self.nazwa


  def przywitajSie(self):
    u'''Powitanie robota.
    Tak, one naprawdę to potrafią.'''
    print 'Melduję się, moi panowie nazywają mnie %s.' % self.nazwa

droid1 = Robot('R2-D2')
droid1.przywitajSie()

droid2 = Robot('C-3PO')
droid2.przywitajSie()
```

Wyjście

```
(Inicjalizacja R2-D2)
Melduję się, moi panowie nazywają mnie R2-D2.
(Inicjalizacja C-3PO)
Melduję się, moi panowie nazywają mnie C-3PO.
```

Objaśnienie

Zdefiniowaliśmy klasę Robot i wygenerowaliśmy dwa obiekty – roboty tej klasy. Nazwa robota jest własnością dotyczącą tylko obiektów opisujących roboty i poza przestrzenią nazw klasy Robot może nie być dostępna, lub oznaczać zupełnie coś innego.

Dziedziczenie

Jedną z najważniejszych korzyści wynikających z programowania obiektowego (nazywanego też obiektowo orientowanego) jest możliwość ponownego wykorzystania już raz napisanego kodu. Osiągamy to poprzez mechanizm dziedziczenia. Dziedziczenie można sobie łatwo wyobrazić jako tworzenie typów pochodnych od istniejących klas.

Załóżmy, że musisz napisać program, który będzie zbierał dane na temat wykładowców i studentów na uczelni. Mają oni pewne wspólne cechy, jak imię, wiek, czy adres. Mają także cechy specyficzne, jak pensja, szkolenia i urlopy dla wykładowców oraz oceny i czesne dla studentów.

Możemy oczywiście stworzyć dwie niezależne klasy dla każdego typu, ale wtedy wspomniane cechy wspólne muszą zostać zdefiniowane w obu tych klasach. Takie postępowanie szybko okaże się nieporęczne.

Możemy oczywiście stworzyć dwie niezależne klasy dla każdego typu, ale wtedy wspomniane cechy wspólne muszą zostać zdefiniowane w obu tych klasach. Takie postępowanie szybko okaże się nieporęczne. Mechanizm dziedziczenia pozwala utworzyć klasę ogólną SchoolMember (członek społeczności szkolnej), z której klasy dla wykładowców i studentów dziedziczą wspólne własności. Dodajemy tylko specyficzne dla nich dane i zachowania. Takie podejscie ma wiele zalet.

1. Jeśli dodamy lub zmodyfikujemy jakąś funkcjonalność w klasie SchoolMember, zmiany te zostaną oczywiście automatycznie uwzględnione przez klasy pochodne. Można na przykład wyobrazić sobie taką sytuację, w której na uczelni wprowadzono wspólne dla wykładowców i studentów karty identyfikacyjne. Wtedy wystarczy, że dodamy odpowiednie pole z numerem karty do klasy SchoolMember, aby ta funkcjonalność pojawiła się zarówno w klasie wykładowców, jak i studentów. Z drugiej strony zmiany w klasach pochodnych nie wpływają na funkcje „przodka" ani pozostałych klas pochodnych.
2. Kolejną zaletą jest fakt, że można traktować obiekty studentów i wykładowców wspólnie jako obiekty jednej klasy - „SchoolMember". Na przykład podczas liczenia wszystkich ludzi związanych z uczelnią przeglądamy obiekty klasy SchoolMember, nie wnikając czy to są studenci, czy wykładowcy. Taka właściwość klas nazywa się polimorfizmem. Polimorfizm w skrócie polega na tym, że zawsze, kiedy oczekiwany obiekt danej klasy, możesz zamiast niego użyć także obiekt klasy pochodnej.
3. Kod zawarty w klasie podstawowej jest wielokrotnie wykorzystujemy. Bez mechanizmu dziedziczenia musielibyśmy pisać ten kod dla każdej klasy osobno. Klasa SchoolMember jest w naszym przypadku klasą podstawową, klasą przodka albo superklasą. Klasy Wykladowca i Student są klasami pochodnymi albo podklasami.

Zobaczmy teraz, jak wygląda wyżej opisany przykład zapisany jako program w Pythonie.

Przykład 8.6: **o_inherit**

```
1  class SchoolMember:
2      u'''Reprezentuje człowieka związanego z uczelnią.'''
3      def __init__(self, imie, wiek):
4          self.imie = imie
5          self.wiek = wiek
6          print '(Inicjalizacja SchoolMember: %s)' % self.imie
7
8      def powiedz(self):
```

```
9           u'''Opowiedz o sobie.'''
10          print 'Imię:"%s" Wiek:"%s"' % (self.imie, self.wiek),
11
12  class Wykladowca(SchoolMember):
13      u'''Reprezentuje wykładowcę.'''
14
15      def __init__(self, imie, wiek, pensja):
16          SchoolMember.__init__(self, imie, wiek)
17          self.pensja = pensja
18          print '(Inicjalizacja Wykladowcy: %s)' % self.imie
19
20      def powiedz(self):
21          SchoolMember.powiedz(self)
22          print 'Pensja: "%d"' % self.pensja
23
24  class Student(SchoolMember):
25      '''Reprezentuje studenta.'''
26      def __init__(self, imie, wiek, oceny):
27          SchoolMember.__init__(self, imie, wiek)
28          self.oceny = oceny
29          print '(Inicjalizacja Studenta: %s)' % self.imie
30
31      def powiedz(self):
32          SchoolMember.powiedz(self)
33          print 'Oceny: "%d"' % self.oceny
34
35  w = Wykladowca('Mrs. Shrividya', 40, 30000)
36  s = Student('Swaroop', 25, 75)
37
38  print # wypisuje pustą linię
39
40  osoby = [w, s]
41  for osoba in osoby:
42      osoba.powiedz() # działa zarówno dla Wykładowców, jak i Studentów
```

Rezultat:

```
(Inicjalizacja SchoolMember: Mrs. Shrividya)
(Inicjalizacja Wykladowcy: Mrs. Shrividya)
(Inicjalizacja SchoolMember: Swaroop)
(Inicjalizacja Studenta: Swaroop)

Imię:"Mrs. Shrividya" Wiek:"40" Pensja: "30000"
Imię:"Swaroop" Wiek:"25" Oceny: "75"
```

Objaśnienie

Aby użyć dziedziczenia, w linijce, w której definiujemy klasę, za jej nazwą zamieszczamy w krotce nazwy klas, z których dziedziczymy. W innym miejscu w kodzie widzimy jawne wywołanie metody __init__ z klasy podstawowej, co było możliwe przez przekazanie jej self. Tym sposobem możemy małym nakładem pracy zainicjalizować tę część klasy pochodnej, która została odziedziczona z klasy podstawowej. Musisz tutaj zapamiętać jedną ważną rzecz: Python nie wywoła automatycznie konstruktora (metody __init__) klasy podstawowej w konstruktorze klasy pochodnej — musisz zrobić to samodzielnie.

Zobaczyliśmy w tym przykładzie, że wywoływanie metod z klasy podstawowej polega na poprzedzeniu ich nazwą klasy podstawowej i kropką oraz przekazaniu im zmiennej self jako pierwszy argument.

Zwróć uwagę na to, że możemy traktować instancje klas Wykladowca oraz Student jak instancje klasy SchoolMember, kiedy używamy metody powiedz z klasy SchoolMember. Zauważ jednak, że wywoływana jest nie metoda powiedz z klasy podstawowej, ale jej wersja z klasy pochodnej, której obiekt aktualnie kryje się pod maską klasy podstawowej. Dzieje się tak dlatego, że Python zawsze rozpoczyna szukanie metod od klasy, z której obiektem w danej chwili ma do czynienia. Dopiero kiedy nie znajdzie żądanej metody, szuka jej w klasach podstawowych w takiej kolejności, w jakiej zostały one zamieszczone w krotce w definicji klasy pochodnej. Jeszcze jedna uwaga co do terminologii — jeśli w krotce z klasami dziedziczonymi znajduje się więcej niż jedna klasa, takie dziedziczenie nazywamy dziedziczenie wielokrotnym albo wielodziedziczeniem.

Metody statyczne

W niektórych językach programowania (na przykład w popularnym języku Java) nie ma możliwości definiowania funkcji poza klasami i obiektami. Nawet najprostsza funkcja nie odwołująca się do niczego poza parametrami musi być wewnątrz jakiejś klasy. Naturalne w świecie Pythona operowanie na obiektach (a nie na klasach) nakazywałoby w takiej sytuacji tworzyć obiekty tylko po to, by wykonać funkcję. Aby tego uniknąć wprowadzono w klasach funkcje statyczne – które istnieją nawet gdy nie ma żadnego obiektu tej klasy. Coś podobnego wprowadzono w Pythonie przy pomocy wbudowanej funkcji staticmethod (metoda statyczna). Zmieńmy nieco powyższy przykład dodając pole do zapamiętanie w klasie (a nie obiekcie) populacji wszystkich robotów danej klasy. Wykorzystamy poprzednią definicję klasy stosując dziedziczenie:

Przykład 8.7: **o_staticmethod**

```
class RobotNG(Robot):
  populacja = 0

  def __init__(self, nazwa):
    Robot.__init__(self,nazwa)
    RobotNG.populacja += 1

  def jakDuzo():
    u'''Wypisuje aktualną populację.'''
    print 'Mamy %d roboty.' % RobotNG.populacja

  jakDuzo=staticmethod(jakDuzo)


droid1 = RobotNG('R2-D2')
droid1.przywitajSie()

droid2 = RobotNG('C-3PO')
droid2.przywitajSie()

RobotNG.jakDuzo()
```

Wyjście:

```
(Inicjalizacja R2-D2)
Melduję się, moi panowie nazywają mnie R2-D2.
(Inicjalizacja C-3PO)
Melduję się, moi panowie nazywają mnie C-3PO.
Mamy 2 roboty.
```

Objaśnienie

Możemy stosować dwa typy pól w obiektach — zmienne klas i zmienne obiektów. Do pierwszych odwołujemy się poprzez nazwę klasy, a do drugich poprzez przedrostek self. Zmana (własność) klasy jest wspólna dla całej klasy. Tu zmienną (własnością) klasy jest populacja, a zmienną obiektów jest nazwa.

Zmienne klasy są dzielone, co oznacza, że są dostępne dla wszystkich instancji danej klasy. Istnieje tylko jedna kopia zmiennej klasy, czyli jeśli jeden obiekt zmieni w jakiś sposób tę zmienną, to zmiana ta będzie widziana również przez wszystkie pozostałe instancje.

Tak więc do zmiennej populacja, która jest zmienną klasy, odnosimy się przez RobotNG.populacja, a nie przez self.populacja. Natomiast do zmiennej obiektu nazwa odnosimy się w metodach tego obiektu przez self.nazwa. Zapamiętaj tę różnicę pomiędzy zmiennymi klas i zmiennymi obiektów. Zapamiętaj też, że zmienna obiektu o takiej samej nazwie, jak zmienna klasy, zasłoni zmienną klasy! Zauważ, że metoda __init__ inicjalizuje instancję klasy Robot nazwą podaną przez argument. Metoda __init__ obiektu pochodnego (RobotNG) zwiększa dodatkowo o 1 licznik populacja. Pokazano także odwołanie do metody (konstruktora) przodka. Robimy to po prostu podając nazwę klasy przodka (zamiast self.). Zauważ też, że wartość self.name jest inna dla każdego obiektu, ponieważ jest to zmienna obiektu.

Zapamiętaj, że do metod i pól tego samego obiektu możesz odnosić wyłącznie za pomocą **self**.

Oprócz __init__, istnieją również inne specjalne metody. Jedną z nich jest __del__, która jest wywoływana w momencie, w którym obiekt umiera, to znaczy kiedy już nigdy nie będzie użyty i zostaje zwrócony systemowi, aby ten mógł zwolnić zajmowaną przezeń pamięć. W tej metodzie możemy zmniejszać na przykład licznik RobotNG.populacja o 1 (przykład tego nie obejmuje)

Metoda __del__ jest uruchamiana, kiedy obiekt już nie jest w użyciu, ale nie ma żadnej gwarancji, kiedy to nastąpi. Jeśli chcesz jawnie ją wywołać, możesz użyć polecenia **del**.

Metoda jakDuzo jest przykładem metody należącej do klasy, a nie do obiektu (nie posiada **self**). Oznacza to, że możemy zdefiniować ją jako classmethod lub staticmethod w zależności od tego, czy chcemy zachować informację, do jakiej klasy ta metoda należy. Ponieważ nie potrzebujemy w naszym programie takiej informacji, użyjemy **staticmethod**.

Dekoratory – czyli nowa magia

Skoro jak powiedziano wcześniej taki styl programowania (sięganie do klas, a nie obiektów) zaciemnia nam program – to czemu do tego wracamy? I jeszcze te nowe dziwne w świecie Pythona „statyczne metody" (staticmethod). Powyższy przykład pozwoli nam Głównie dlatego, by zrozumieć działanie dekoratorów. Jak wcześniej to opisano, Python niejawnie tłumaczy wywołanie metody obiektu na wywołanie funkcji zdefiniowanej w klasie, wstawiając referencję obiektu do parametru self. Zamiast więc złożonej konstrukcji Klasa.metoda(obiekt) mamy eleganckie obiekt.klasa(). Podany wcześniej przykład z robotami pokazuje, że także nie zalecane modyfikowanie definicji (klasy) może mieć sens. Aby więc ukryć paskudne szczegóły, tu także pojawiła się magia w postaci dekoratorów. Tu mamy możliwość zastosowania prostego dekoratora:

```
1  @staticmethod
2  def jakDuzo():
3    u'''Wypisuje aktualną populację.'''
4    print 'Mamy %d robotów.' % Robot.populacja
```

Dekoratory można sobie wyobrazić jako skróty do wywołania jawnego polecenia, jak to widzieliśmy w przykładzie. Ogólnie zapis:

```
@mydecorator
def myfunc():
  pass
```

można traktować jako równoważnik równoważnik do zapisu:

```
def myfunc():
  pass
myfunc = mydecorator(myfunc)
```

Prosty przykład dekoratora

Przykład 8.8: **o_decorator**

```
def helloworld(ob):
  print "Hello world"
  return ob

@helloworld
def myfunc():
  print "my function"

myfunc()
```

Wyjście:

```
Hello world
my function
```

Objaśnienie:

Dodanie oznaczenie dekoratora @helloworld powoduje, że wywołanie myfunc() zostanie poprzedzone funkcjonalnością z helloworld. Nieco bardziej złożony przykład dotyczy funkcji z parametrami.

Przykład 8.9: **sumator**

```
def dekorator(f):
  def nowa_funkcjonalnosc(*args, **kwds):
    print('Nazwa funkcji:', f.__name__)
    return f(*args, **kwds)

  return nowa_funkcjonalnosc


@dekorator
def funkcja1(par):
  print(par)

funkcja1('OK1')

# równoważny kod bez użycia notacji dekoratorów
def funkcja2(par):
    print(par)

f=dekorator(funkcja2)
f('OK2')
```

Wyjście

```
('Nazwa funkcji:', 'funkcja1')
OK1
('Nazwa funkcji:', 'funkcja2')
OK2
```

Objaśnienie

Użycie funkcja2 wyjaśnia działanie dekoratora przy funkcja1. Należy zwrócić jednak uwagę na dwa dodatkowe elementy:

1. Wewnątrz funkcji dekorator zdefiniowano inną funkcję i przekazano ją jako parametr. To jest rzadko stosowane - poza właśnie definicją dekoratorów.
2. Nieco częściej można spotkać alternatywny zapis parametrów: *args, **kwds. Oznacza to dowolną listę parametrów. Opisano to w poprzednim. Więcej informacji o dekoratorach znajdziesz na stronie: http://www.rwdev.eu/articles/decorators

Interfejsy i wtyczki

Powszechność występowania obiektów świata skłania do stosowania języków obiektowych do sterowania nimi. Robimy to za pomocą różnych interfejsów i wtyczek.

1. Interfejsem – czyli inaczej złączem jest specyfikacja połączenia. Coś w rodzaju instrukcji obsługi. Gdy ta obsługa ma się odbywać z użyciem programu – instrukcja przyjmuje zazwyczaj postać opisu obiektu (obiektów). Szczegółowa dokumentacja opisująca jak taki obiekt zbudować, nazywa się API (z angielskiego Application Programming Interface). Przykładowo encyklopedia internetowa ma swoje API zdefiniowane na stronie: https://www.mediawiki.org/wiki/API:Main_page. Jak w większości popularnych interfejsów, programiści Pythona już z tej instrukcji skorzystali. Nie musimy więc wcale jej czytać. Wystarczy krótki opis obiektu, by go użyć. Najpierw instalujemy w swoim systemie:

```
pip wikipedia,
```

teraz już możemy sięgać do Wikipedii z programu:

```
import wikipedia
wikipedia.summary('Star Treck')
```

1. Wtyczki (ang. plugin) to bardziej złożony sposób korzystania z obiektów. Chodzi mianowicie o rozszerzenie ich funkcjonalności. W tym wypadku uruchamiamy program tylko wewnątrz innego programu. Na przykład popularny program graficzny Gimp ma swoje API. Napiszmy prostą wtyczkę, która przypomina nasze ulubione kolory.

Stwórzmy program mojekolory.py

```
#!/usr/bin/env python

from gimpfu import *

def MojeKolory(img, drawable):
    pdb.gimp_context_set_background((50,255,255))
    pdb.gimp_context_set_foreground((0, 8, 255))


register(
"python-fu-kolory",
"Ustaw Kolory",
```

```
13  "Standardowe Kolory",
14  "JW",
15  "Galicea",
16  "2016",
17  "<Image>/Filters/Moje Kolory",
18  "*",
19  [],
20  [],
21  MojeKolory)
22
23  main()
```

Program należy skopiować w miejsce przeznaczone na wtyczki – zgodnie z instrukcją: https://www.gimpuj.info/inde
W tym programie funkcja MojeKolory sięga do obiektu pdb, który reprezentuje ustawienia Gimpa i wywołuje metody zmiany kolorów. Reszta powyższego programu to rejestracja wtyczki – zmiana menu. W podmenu Filtry pojawi się funkcja „Standardowe Kolory” (dostępna w trakcie edycji obrazu). Jej wybranie powoduje uruchomienie procedury MojeKolory, która ustawia domyślne kolory tła i pędzla.

Przetwarzanie danych

Struktury danych takie jak **listy** są bardzo użyteczne i wygodne w użyciu. Mają jednak pewną wadę. Są przechowywane w pamięci operacyjnej komputera. Pamięć ta ma ograniczoną pojemność a na dodatek jest ulotna. Jeśli więc zostanie wyłączone zasilanie, albo program zostanie usunięty z pamięci – dane znikną wraz z nim. W niniejszym rozdziale zajmiemy się metodami trwałego zapisu danych.

Pliki

Pliki to dane zapisane na dysku lub innym urządzeniu. Dostęp do plików uzyskuje się w Pythonie poprzez standardową (dostępną bez importu dodatkowych modułów) klasę **file**. Utworzenie obiektu tej klasy wiąże się z **otwarciem** pliku. Otwarcie pozwala na zapis i/lub odczyt danych. Plik otwieramy wywołując funkcję **open** podając w parametrach nazwę pliku. Drugi (opcjonalny) parametr wskazuje, czy zamierzamy do pliku pisać (‘**w**’), czy z niego czytać (‘**r**’). Funkcja **open** zwraca się do systemu operacyjnego aby zezwolił na dostęp (odczyt / zapis) do pliku. Gdy to się uda, zwracany jest obiekt klasy **file**.

Pliki najczęściej zawierają teksty i przetwarza się je wierszami.

Przykład 9.1: **pliki**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

wierszyk = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
'''

f1 = open('wierszyk.txt', 'w')
            # 'w' oznacza, że będziemy pisać do pliku
f1.write(wierszyk)
f1.flush()       # zapisanie zmian w pliku na dysk

f2 = open('wierszyk.txt')
            # brak drugiego argumentu oznacza 'tylko odczyt'
```

```
18 for wers in f2:
19   print(wers)
```

Rezultat:

```
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

Objaśnienie:

1. Tworzymy plik wierszyk.txt otwierając go w trybie do zapisu (z argumentem ‘w’). Jeśli plik o takiej nazwie jeszcze nie istniał, to zostaje on utworzony z pustą zawartością. Jeśli już istniał, to jego zawartość zostaje skasowana (jeśli chcemy dopisać treść na końcu nie kasując zawartości, to zamiast parametru ‘w’ powinniśmy użyć parametru ‘a’). Do tego pliku zapisujemy tekst podzielony na wiersze.
2. Następnie otwieramy ten sam plik ponownie, tym razem w trybie do odczytu. Następnie linijka po linijce odczytujemy plik i wypisujemy każdą linijkę.

Program jest bardzo prosty, ale jest tu parę drobiazgów na które warto zwrócić uwagę:

1. Zmienna wierszyk jest wielowierszowym ciągiem znaków. W Pythonie takie teksty tworzymy przez użycie potrójnego apostrofu (‘’’) jako ogranicznika.
2. Bezpośrednio po otwarciu tego napisu jest użyty znak kontynuacji ‘\’. Gdyby go nie było, napis zawierałby dodatkową pustą linijkę na samym początku.
3. Zapis danych na dysk nie następuje natychmiast, ale dopiero wtedy, gdy uzbiera się ich większa ilość. Dlatego na końcu powinno się wykonać operację odwrotną do otwarcia: close. Zapisuje ona wszystkie dane na dysk i kasuje obiekt pliku. Jeśli chcemy zacząć odczytywać ten plik bez zamykania - musimy się upewnić, że nasze zmiany naprawdę znalazły się na dysku. Służy do tego metoda **flush**.
4. Gdybyśmy nie wywołali flush, to zmiany w pliku i tak zostałyby zapisane automatycznie najpóźniej w momencie zakończenia programu. Gdy program kończy swoje działanie, wszystkie zmiany są zapisywane i pliki są zamykane przez system operacyjny.
5. Iteracja po obiekcie file oznacza iterację po linijkach (for otrzymuje kolejne linijki).

Jeśli chcemy zapisać inne niż tekst dane, możemy posłużyć się funkcją pakowania danych (**pack**) z modułu **struct**. Zamienia ona tekst na dane binarne (w takiej postaci jak przechowywane są w pamięci komputera). Każda dana zajmuje tyle miejsca (w bajtach), ile podano w tabelce opisującej tą funkcję https://docs.python.org/3/library/struct.html?highlight=struct.pack[32]:

[32] https://docs.python.org/3/library/struct.html?highlight=struct.pack

Przykład 9.2: **bpliki**

```
#!/usr/bin/env python

import os

f1=open('plik1.txt','w')
buf='3334'
f1.write(buf)
f1.close()

from struct import *
f2=open('plik2.dat','wb')
buf=pack('hh', 33,34)
f2.write(buf)
f2.close()

fin = open("plik2.dat", "rb")
bufin = fin.read(4)__
print(unpack('hh',bufin))
fin.seek(2)
print(unpack('h',fin.read(2)))

fin.close()
```

Wyjście

```
(33, 34)
```

Objaśnienie

1. Funkcja **pack** tworzy bufor danych, który może być zapisywany do pliku binarnego (funkcją write dla obiektu file) – identycznie jak do plików tekstowych zapisuje się napisy (string). Pierwszy parametr funkcji **pack** zawiera określenie jakiego rodzaju dane mają być pakowane – zgodnie ze wskazaną wcześniej tabelką (h oznacza małą liczbę całkowitą – short integer). Po nim następują dane do upakowania.
2. Odczytując dane binarne musimy podać ilość danych do odczytania. Następny odczyt odbywa się od miejsca, gdzie skończył się poprzedni. Mówi się czasem o wskaźniku pliku – jako wskazaniu na miejsce odczytu/zapisu. Można dzięki temu do jednego pliku pisać i czytać wiele danych binarnych. Taki sekwencyjny (jeden po drugim) odczyt danych może dla dużych plików być nieefektywny (gdy chcemy odczytać dane nie zapisane na samym początku pliku). Dlatego dla plików na dysku wprowadzono przesuwanie wskaźnika funkcją seek. W naszym przykładzie seek(2) oznacza przejście do 3-go bajtu liczonego od początku pliku.

3. Literka ‘b’ dodana do parametru określającego tryb otwarcia pliku (‘wb’ zamiast ‘w’ oraz ‘rb’ zamiast ‘b’) oznacza, że chcemy zapisywać / odczytywać dane binarne. W przypadku Pythona 2.x w zasadzie ta różnica jest bez znaczenia (ale jest akceptowana), ale w Pythonie 3 jest istotna.

Pickle

Dostęp do plików binarnych choć efektywny, może powodować trudne do zlokalizowania błędy. Wystarczy drobna pomyłka w wyliczeniu ilości bajtów lub ich oznaczenia w funkcji pack/unpack – i otrzymujemy błędne dane. Dlatego standardowa biblioteka Pythona zawiera moduł **pickle** (konserwacja, utrwalanie), który przejmuje za nas tego typu działania. Służy on do zapisywania i odczytywania dowolnych obiektów Pythona. Tego typu obiekty są nazywane trwałymi (ang. *persistent*) a plik je przechowujący pamięcią trwałą (persistent storage). Zakończenie programu nie powoduje bowiem ich zniknięcia.

Przykład 9.3: **pickling**

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import pickle
5
6  # the name of the file where we will store the object
7  trwala_lista_zakupow = 'lista_zakupow.data'
8  # lista rzeczy do kupienia
9  lista_zakupow = ['jabłka', 'ziemniaki', 'pomidory']
10
11 # zapisanie do pliku
12 f = open(trwala_lista_zakupow, 'wb')
13 pickle.dump(lista_zakupow, f)
14 f.close()
15
16 # usunięcie zmiennej 'nie-trwałej'
17 del lista_zakupow
18
19 # odzyskanie zmiennej
20 f = open(trwala_lista_zakupow, 'rb')
21 lista_zakupow = pickle.load(f)
22 for rzecz in lista_zakupow:
23     print(rzecz)
```

Wyjście

```
jabłka
ziemniaki
pomidory
```

Objaśnienie:

1. Aby zapisać 'utrwalane' dane do pliku, otwieramy go – podobnie jak w poprzednim przykładzie w trybie do zapisu binarnego ('wb' = *write – binary*).
2. Aby odczytać takie dane z pliku, otwieramy plik w trybie do odczytu. Funkcja **pickle.load()** pozwala nam odczytać dokładnie to co zapisaliśmy.

Pliki i wyrażenie with

Konieczność zamykania plików to coś, o czym programista powinien pamiętać. Często jednak jest to pomijane (jak w poprzednim przykładzie), bo przecież i tak z chwilą zakończenia programu wszystko zostanie zamknięte. Nie jest to uniwersalne i godne polecenia rozwiązanie. Dlatego wprowadzono instrukcję **with** i wyposażono obiekty otwierane (takiej jak **file**) w funkcjonalność pozwalającą na automatyczne ich zamykanie. Obiekt jest zamykany z chwilą zakończenia bloku **with**.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  with open('wierszyk.txt') as plik:
5    for wers in plik:
6      print(wers)
```

Objaśnienie:

1. Dzięki użyciu funkcji open razem z wyrażeniem with — nie przejmujemy się zamykaniem pliku, gdyż with open robi to automatycznie.
2. Instrukcja with została wbudowana w Pythonie w wersji 2.6. W Pythonie 2.5 with musi być najpierw zaimportowane z modułu **__future__**:

```
from __future__ import with_statement # Niepotrzebne w Pythonie 2.6 i wyższych
```

1. Instrukcja with zawsze wywołuje funkcję **file.__enter__** przed rozpoczęciem wykonania swojego bloku oraz – na koniec funkcję **file.__exit__**.

Bazy danych

Upowszechnienie się baz danych sprawiło, że nie musimy implementować szczegółów operacji zapisu i odczytu danych, ale możemy je zlecić innym programom: serwerom baz danych. Zlecenia te są formułowane w języku SQL[33], którego najprostsze polecenie ma kształt:

```
select * from tabela
```

Oznacza to odczytanie wszystkich danych (*) z tabeli o nazwie **tabela**.

W internecie można znaleźć wiele kursów języka SQL. Jako całe wprowadzenie wystarczy jednak poniższy prosty (choć kompletny) przykład.

Przykład 9.4: **sqlite**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import sqlite3

def connect_db():
    # przylaczenie bazy danych
    db = sqlite3.connect('test.db')
    return db

def ini_db():
    try:
        db = connect_db()
        cur = db.cursor()
        cur.executescript('''\
CREATE TABLE "user" (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  "username" TEXT,
  "password" TEXT
);
        ''')
        db.commit()
    except sqlite3.Error as e:
        print("DB error:", e.args[0])
    except Error as e:
```

[33] https://pl.wikipedia.org/wiki/SQL

```
27          print("rror:", e.args[0])
28
29  def new_user(db, name, password):
30      try:
31          cursor = db.cursor()
32          cursor.executescript("insert into user(username, password) values ('%s',\
33  '%s')" % (name, password))
34          # nie powoduje błędu z O''Connor:
35          #   cursor.execute("insert into user(username, password) values (?,?)" ,\
36   (name, password))
37          db.commit()
38      except sqlite3.Error as e:
39          print("DB error:", e.args[0])
40          print('nieudane  wstawienie %s ', name)
41
42  ini_db()
43  db=connect_db()
44  new_user(db, 'Tomek','haslotomka')
45  new_user(db, 'Tomek','haslotomka')
46  new_user(db, "O'Connor",'haslo2')
47  new_user(db, 'Romek','hasloromka')
48
49  cursor = db.cursor()
50  for (id, username, password) in cursor.execute('select id, username, password fr\
51  om user'):
52      print('%s %s:%s' % (id, username, password))
```

Wynik:

```
('DB error:', 'near "Connor": syntax error')
('nieudane  wstawienie %s ', "O'Connor")
1 Tomek:haslotomka
2 Romek:hasloromka
```

Objaśnienie:

1. Istnieje wiele różnych baz danych. Powyższy przykład dotyczy najprostszej z nich: SQLite. Dane są pamiętane w przypadku tej bazy w jednym pliku. W przypadku bardziej zaawansowanych baz danych komunikacja odbywa się za pośrednictwem sieci. Dla SQLite potrzebny jest tylko odpowiedni moduł – w przypadku Pythona jest to sqlite3.

2. Połączenie z bazą danych (funkcja connect) ustala zazwyczaj jedynie parametry komunikacji. W przypadku SQLite polecenie sqlite3.connect(‘test.db’) tworzy bazę – o ile wcześniej ona nie istniała.
3. Komunikacja z bazą odbywa się z użyciem języka SQL. Polecenia nie są przesyłane wprost poprzez obiekt połączenia (db w powyższym przykładzie), ale przez obiekt kursora (ang. cursor). Umożliwia to stworzenie wielu kursorów dla jednej bazy – na przykład dwa kursory do przeglądania równolegle dwóch tabel.
4. Do wysłania polecenia SQL można użyć funkcji executescript lub execute. W tym drugim przypadku jako wynik funkcji dostajemy rezultat zapytania w postacie generowanych wierszy.
5. Polecenie SQL ‘CREATE TABLE “user” … ‘ tworzy tabelę o nazwie „user”. Polecenie ‘insert into ….’ wstawia wiersze do tej tabeli. Natomiast ‘select …’ odczytuje wstawione dane.
6. Funkcja commit() w obiekcie połączenia z bazą powoduje zakończenie zapytania i zapisanie danych na dysku.

[v3]

Pisanie niezawodnego kodu

Wyjątki, czyli przewidywanie niespodziewanego

Rozważmy proste wywołanie polecenia **print**. Co się stanie, gdy przypadkiem zapiszemy **print** jako **Print**? W tym wypadku Python zgłasza błąd nazwy polecenia.

```
>>> Print (1, 2, 3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
>>> print (1, 2, 3)
(1, 2, 3)
```

Python wyświetla rodzaj błędu (**NameError**) oraz miejsce, w którym ten błąd wystąpił. Użycie identyfikatora **NameError** jest związane z mechanizmem obsługi sytuacji wyjątkowych do których należy wystąpienie błędu w programie. Mechanizm ten nazywa się **wyjątkami**.

NameError jest nazwą wyjątku, jaki udało się zidentyfikować. Jest to identyfikator klasy pochodnej wobec Exception.

Inny przykład wyjątku możemy uzyskać wprowadzając błędne dane. Funkcja raw_input zakłada, że jakieś dane się pojawią.
Jeśli zamiast wprowadzania danych naciśniemy **Ctrl-d** – będzie to oznaczało koniec danych.

```
>>> s = raw_input('Wpisz coś ---> ')
Wpisz coś ---> Traceback (most recent call last):
File "<stdin>", line 1, in <module>
EOFError
```

Python zgłasza błąd typu EOFError, co oznacza, że znalazł symbol *końca pliku* (który jest reprezentowany przez **ctrl-d**) w miejscu, w którym on nie powinien występować (**EOF** = *End Of File* = koniec pliku).

Obsługa wyjątków

Możemy zaprogramować własną obsługę wyjątków używając instrukcji **try**...**except**....

Po słowie kluczowym except umieszczamy kod, który zostanie wykonany, gdy w bloku instrukcji po słowie kluczowym try nastąpi wyjątek.

Przykład 10.1: **try_except**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Nazwa pliku: try_except.py

try:
    tekst = raw_input('Wpisz coś ---> ')
except EOFError:
    print 'Dlaczego użyłeś znaku końca pliku?'
except KeyboardInterrupt:
    print 'Przerwałeś operację.'
else:
    print 'Wpisałeś %s' % tekst
```

Rezultat:

```
$ python try_except.py
Wpisz coś ---> Dlaczego użyłeś znaku końca pliku?
$ python try_except.py
Wpisz coś ---> ^CPrzerwałeś operację.
$ python try_except.py
Wpisz coś ---> tekst nie wywołujący błędów
Wpisałeś tekst nie wywołujący błędów
```

Za pierwszym razem naciśnięto **ctrl-d**, za drugim **ctrl-c**.

Objaśnienie:

1. W razie wystąpienia wyjątku w bloku instrukcji po słowie **try**, resztatego bloku jest pomijana. Python próbuje ustalić jaki rodzaj wyjątku wystąpił i przechodzi do odpowiedniej dla niego sekcji obsługi – po słowie **except**. W przypadku, gdy rodzaj błędu nie zostanie określony, wykonywany jest ostatni blok **except** – który nie jest oznaczony nazwą wyjątku.
2. Dla każdego **try** musi być zdefiniowane przynajmniej jedna sekcja wyjątków **except**.
3. Gdy nie obsłużymy błędu za pomocą except, wtedy Python domyślnie zatrzymuje program i wyświetla komunikat o błędzie – tak jak to wcześniej pokazano.
4. Blok **else** w przypadku try działa podobnie jak przy **if** oraz **while** – wykonywany jest wówczas, gdy nie wystąpił żaden wyjątek.

Zgłaszanie wyjątków

Wyjątki zdarzają się nie tylko z powodu nieprzewidzianych okoliczności. Programista może przewidzieć sytuację wyjątkową i po jej stwierdzeniu zgłosić wyjątek instrukcją **raise**.

Przykład 10.2: **try_raise**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

class WyjatekKrotkiegoWpisu(Exception):
    '''Klasa wyjątku zdefiniowana przez użytkownika.'''
    def __init__(self, dlugosc, conajmniej):
        Exception.__init__(self)
        self.dlugosc = dlugosc
        self.conajmniej = conajmniej

try:
    tekst = raw_input('Wpisz coś ---> ')
    if len(tekst) < 3:
        raise WyjatekKrotkiegoWpisu(len(tekst), 3)
    # Pozostałe polecenia mogą tu być normalnie wpisane.
except EOFError:
    print 'Dlaczego użyłeś znaku końca pliku?'
except WyjatekKrotkiegoWpisu, wkw:
    print 'WyjatekKrotkiegoWpisu: Wpis miał długość %d, wymagane co najmniej %d.\
'\
        % (wkw.dlugosc, wkw.conajmniej)
else:
    print 'Żaden wyjątek nie został zgłoszony.'
```

Rezultat:

```
Wpisz coś ---> a
WyjatekKrotkiegoWpisu: Wpis miał długość 1, wymagane co najmniej 3.
$ python zglaszanie.py
Wpisz coś ---> abc
Żaden wyjątek nie został zgłoszony.
```

Objaśnienie

Jak już wcześniej była mowa – dla każdego wyjątku tworzony jest obiekt klasy pochodnej w stosunku do Exception. Możemy oczywiście stworzyć własną klasę wyjątków. W tym przykładzie stworzyliśmy klasę o nazwie **WyjatekKrotkiegoWpisu**. Tworzenia własnych klas zapewnia nie tylko rozróżnianie wyjątków, ale można stworzyć dodatkowe własności w klasie wyjątku do przechowywania związanych z obsługiwanym zdarzeniem wartości. W naszym przykładzie jest to **dlugosc**, czyli długość wpisanego tekstu oraz **conajmniej**, czyli minimalna długość, jakiej program oczekuje.

try...finally...

Czasem trzeba wykonać jakieś działania niezależnie od tego, czy nastąpił wyjątek. Na przykład chcemy zawsze zamknąć otwarty plik. Służy do tego blok instrukcji try umieszczany po słowie finally.

Przykład 10.3: try_finally

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Nazwa pliku: try_finally.py

import time

try:
    plik = open('wierszyk.txt')
    while True:
        wers = plik.readline()
        if len(wers) == 0:
            break
        print wers,
        time.sleep(2) # Żeby program się na chwilę zatrzymał.
except KeyboardInterrupt:
    print '!! Przerwałeś odczytywanie pliku.'
finally:
    plik.close()
    print '(Sprzątanie: Zamknięto plik)'
```

Rezultat:

```
Programming is fun
When the work is done
if you wanna make your work also fun:
^C!! Przerwałeś odczytywanie pliku.
(Sprzątanie: Zamknięto plik)
```

Objaśnienie

Wprowadzenie opóźnienia po odczytaniu każdej linijki (time.sleep) umożliwia naciśnięcie **ctrl-c** i przerwanie działania programu jeszcze w czasie odczytu.

Wyjątek KeyboardInterrupt (przerwanie z klawiatury) zostanie wyświetlony i program zakończy swoje działanie. Jednakże tuż przed końcem programu blok **finally** umożliwia poprawne zamknięcie pliku.

Jest to rozwiązanie alternatywne do opisanej w poprzednim rozdziale instrukcji **with**.

Zostawianie śladów

Samo wyświetlenie komunikatu może nie wystarczyć dla zdiagnozowania ewentualnych problemów. Jeśli na przykład wyjątek pojawi się w złożonym systemie, który musi tolerować problemy (na przykład serwer www) – warto jednak by ślad po tych problemach pozostał do późniejszej analizy. Służy do tego system logowania (dziennik) błędów. W Pythonie zaimplementowano go w module logging.

Poniżej nieco bardziej złożony przykład pokazuje raz jeszcze obsługę błędów, ale tym razem z zapisywaniem problemów do loga.

Przykład 10.4: **logging**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

class BlednyKodPocztowy(Exception):

  def __init__(self, msg):
    Exception.__init__(self)
    self.msg = msg

class KodPocztowy():
  wartosc = '00-000'

  def set(self,kod):
    print('Proba ustawienia %s' % kod)
    if (len(kod)<>6):
      raise BlednyKodPocztowy('Kod ma miec 6 znaków!')
    if (kod[2]<>'-'):
      raise BlednyKodPocztowy('Poprawny format kodu: 99-999')
    self.wartosc=kod

kod=KodPocztowy()
ikod=0

import logging

# bez tego dziala jak print na konsole
logging.basicConfig(filename='example.log',level=logging.DEBUG)

try:
  kod.set('22-22')
```

```
except BlednyKodPocztowy as e:
  logging.warning(e.msg)
except:
  logging.warning('Jakis blad!')

try:
  kod.set('22-222')
  print('Wartosc liczbowa kodu %s ' % kod.wartosc)
  ikod=int(kod.wartosc)
except BlednyKodPocztowy as e:
  logging.warning(e.msg)
except:
  logging.warning('Inny blad!')
```

Objaśnienie

1. W programie zaimplementowano klasę KodPocztowy do przechowywania kodów pocztowych. Zapamiętywanie kodu zapewnia metoda set. Sprawdza ona poprawność kodu i zgłasza wyjątek. Zdefiniowano do tego celu własny wyjątek o nazwie BlednyKodPocztowy. Utworzono także obiekt kod klasy KodPocztowy. Ten fragment programu nie wyróżnia się niczym w stosunku do poprzednich przykładów dotyczących wyjątków.
2. Różnica polega na tym w jaki sposób program obsługuje wyjątki. Do tego celu użyty został mechanizm logowania błędów – logging. Funkcja logging.basicConfig ustawia nazwę pliku do którego będą zapisane błędy, oraz poziom śledzenia (w tym wypadku najwyższy – DEBUG).
3. Wywołanie logging.warning(komunikat) powoduje zapisanie komunikatu do późniejszej analizy.

Automatyczne dokumentowanie

Python posiada mechanizm zwany opisami dokumentującymi określany skrótem **docstrings** od angielskiej nazwy (*documentation strings*). Jest to potężne narzędzie, którego warto często używać, gdyż pomaga w lepszym zrozumieniu działania programu. Pamiętaj, **że program piszesz raz, ale czytany i analizowany będzie wielokrotnie**! Mechanizm docstring w Pythonie może być wykorzystywany nawet w trakcie eksploatacji, gdyż potrafi on uzyskać na przykład opis uruchomionej funkcji!

Przykład 10.5: **docstring**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def printMax(x, y):
    '''\
Wypisuje maksymalną liczbę spośród dwóch podanych.

Obydwie wartości muszą być liczbami całkowitymi.'''
    x = int(x) # Zmienia na liczby całkowite, jeżeli to możliwe.
    y = int(y)

    if x > y:
        print x, 'to maksimum'
    else:
        print y, 'to maksimum'

printMax(3, 5)
print printMax.__doc__
#help(printMax)
```

Rezultat:

```
5 to maksimum
Wypisuje maksymalną liczbę spośród dwóch podanych.

Obydwie wartości muszą być liczbami całkowitymi.
```

Objaśnienie:

Opis w postaci łańcucha znaków w pierwszym wierszu funkcjito jej DocString (opis dokumentujący). Taki sam opis można zastosować do modułów i klas!

Zaleca się (choć nie jest to obowiązkowe), aby opis **docstring** zaczyna... się dużą literą, pierwszy wiersz był zakończony kropką, po nim następował wiersz pusty, a dalej dokładniejsze wyjaśnienia.

Opis (docstring) jest dostepny za pośrednictwem własności __doc__ (uwaga na podwójne podkreślenia). W Pythonie *wszystko* jest obiektem - także nawet funkcje – dlatego można definować własności funkcji (takie jak __+doc__).

Zamiast odwołania do __doc__ można używać funkcji **help()**. W tym wypadku **help(printMax)**.

Opis nie jest wówczas jedynie wyświetlany, ale można go przeglądać. **Uwaga!** Z przeglądania tego opisu wychodzi się naciskając klawisz q.

Kodowanie oparte o testy

Kodowanie sterowane testami polega na tym, że najpierw przygotowujemy test fragmentu programu, a później dopiero kodujemy ten fragment. Test dzieli się na funkcjonalne i jednostkowe. Test funkcjonalny sprawdza działanie funkcji programu z zewnątrz – z perspektywy użytkownika. Pozwala stwierdzić czy program działa zgodnie ze specyfikacją. Testy jednostkowe natomiast sprawdzają działanie z perspektywy programisty – od środka. Gdy program nie działa poprawnie – pomagają precyzyjnie ustalić przyczynę. Ta strategia tworzenia oprogramowania jest stosunkowo młoda. W niniejszym podręczniku jedynie sygnalizujemy jeden ze sposobów realizacji tej idei (obszerniejszy przykład można znaleźć na stronie: http://www.python.rk.edu.pl/w/p/programowanie-oparte-o-testy/[34]).

Wykorzystamy obiekt szyfrowania z jednego z poprzednich przykładów.

Przykład 10.6: **tdd**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# przykłady z książki https://leanpub.com/pyprog
class TSzyfrant:

def __init__(self,klucz):
    self.klucz=klucz

def szyfruj(self, liczba):
    return (liczba + self.klucz) % 256

def odszyfruj(self, liczba):
    return (256 + liczba - self.klucz) % 256

def TestFunkcjonalny():
szyfrant=TSzyfrant(88)
wiadomosc = int(raw_input('Podaj liczbę: '))
szyfr=szyfrant.szyfruj(wiadomosc)
print('Kod=%s' % szyfr)
print('Odszyfrowano=%s' % szyfrant.odszyfruj(szyfr))

import unittest
from random import randrange

class TestyJednostkowe(unittest.TestCase):

```

[34] http://www.python.rk.edu.pl/w/p/programowanie-oparte-o-testy/

```
27  def setUp(self):
28      """Call before every test case."""
29      self.szyfrant = TSzyfrant(88)
30
31  def test1(self):
32      msg=randrange(255)
33      for n in range(10):
34      szyfr = self.szyfrant.szyfruj(msg)
35      msg2 = self.szyfrant.odszyfruj(szyfr)
36      assert  msg == msg2
37
38  if __name__ == '__main__':
39  TestFunkcjonalny()
40  unittest.main() # run all tests
```

Objaśnienie

- Test funkcjonalny w powyższym przykładzie to po prostu dodatkowa funkcja w której wykonuje się obliczenia dla przykładowych danych.
- Nieco bardziej złożone jest pisanie testów jednostkowych. Używane w nich jest polecenie assert, które sprawdza poprawność wykonanych obliczeń. Po słowie kluczowym assert podaje się warunek. Gdy warunek nie jest spełniony – wywoływany jest wyjątek. Dodatkowo po warunku można podać komunikat przesyłany do wyjątku.
- W kodowaniu sterowanym testami zaczynamy od napisania testu, zastępując wywoływane funkcje „zajawkami” (nagłówek + słowo kluczowe pass). Test oczywiście nie przejdzie póki nie wykonamy poprawnej implementacji funkcji.

[v5]

Przepływ danych i sterowania

Zdobyliśmy już całkiem sporo informacji na temat programowania w Pythonie. W obecnym rozdziale spojrzymy na to zagadnienie z nieco innej strony. Zastanowimy się jak w Pythonie zorganizowany jest przepływ danych i sterowania oraz interakcja z otoczeniem. Tu nie ma jednej uniwersalnej metody. Zależnie od środowiska w jakim ma działać nasz program oraz celów w jakim on powstaje, musimy dobrać odpowiednie rozwiązanie. Najczęściej można przy tym skorzystać z czegoś, co inni wymyślili przed nami. Przegląd najważniejszych idei znajdziesz w modularnym kursie programowania http://otwartaedukacja.pl/programowanie/[35]. Poniżej wykorzystamy zawartą tam klasyfikację, aby pokazać jak różne mogą być zastosowania Pythona.

W tym rozdziale najważniejsza jest tak zwana wiedza „miękka" - czyli idee i rozwiązania stosowane przez programistów. Pozwoli to nie tylko uporządkować i utrwalić pewne pojęcia, ale też lepiej zrozumieć sens tego co wykonuje program. Pojawią się jednak także pewne fundamentalne zagadnienia, które nie były wcześniej omawiane (posługiwanie się konsolą, programy systemowe, obsługa błędów, graficzny interfejs użytkownika). To są obszerne zagadnienia, których pełne omówienie wykracza poza zakres tego podręcznika. Poznajemy je dogłębnie wtedy, gdy musimy rozwiązać konkretne zadanie. Tutaj zostaną one pobieżnie omówione na tyle, aby (miejmy nadzieję) wiedzieć gdzie szukać i przy okazji wyjaśnić ideę różnego typu przepływów informacji w programach.

Co to jest sterowanie?

Interpreter Pythona wykonuje kolejne instrukcje programu. Wskazanie na aktualnie wykonywaną instrukcję nazywa się potocznie **sterowaniem**. Program wykonuje kolejne instrukcje w sposób sekwencyjny – tak jak zostały zapisane. Poznaliśmy już wcześniej zmiany sterowania związane z instrukcjami warunkowymi, pętlami, wywołaniem procedur i własności obiektów.

Wykonywane instrukcje mogą zmieniać dane (zmienne). Sterowanie możemy śledzić wykonując program krok po kroku przy pomocy debuggera (funkcjonalność większość IDE – na przykład PyScriptera dla Windows). Debuggery mają także możliwość śledzenia tego jak zmienia się zawartość zmiennych (dane).

1.Konsola Pythona

Konsola Pythona to program, który odczytuje instrukcje w jakimś języku programowania i je wykonuje. Stan w jakim znajduje się komputer może zależeć od wcześniej wykonanych instrukcji jedynie wówczas, gdy zmieniały one jakieś dane (zmienne).

[35] http://otwartaedukacja.pl/programowanie/

W przypadku Pythona sprawa jest banalnie prosta. Wpisywane przez nas instrukcje (zob. rozdział „Python jako kalkulator) interpreter czyta i wykonuje po naciśnięciu klawisza Enter.

A jak jest z blokami instrukcji?

Uruchom interpreter Pythona poleceniem:

```
python
```

i wypróbuj:

```
>>> liczby=[1,4,5,6,7]
>>> for n in liczby:
...   print(n)
...

1
4
5
6
7
>>>
```

Po wpisaniu wiersza zaczynającego instrukcję **for** (a kończącego się dwukropkiem), interpreter nic nie robi – czeka na pusty wiersz. Wpisujemy więc instrukcje drukowania (print) i kończymy pustym wierszem (Enter na początku). Zwróć uwagę na spacje przed słowem **print!**

Interpreter traktuje ciąg wprowadzanych znaków jako jeden strumień, kończący się naciśnięciem klawiszy **Ctrl+D** (przy wciśniętym Ctrl naciśnij D). Napotkanie takich klawiszy kończy przetwarzanie.

Czy po każdej instrukcji interpreter wraca do stanu początkowego? Nie - jeśli ma nie zakończoną instrukcję (jak for w powyższym przykładzie). Poza tym pozostaje „efekt uboczny” w postaci zmiennej „liczby”. Zmienną tą możemy usunąć poleceniem **del**:

```
>>> liczby
[1, 4, 5, 6, 7]
>>> del liczby
>>> liczby
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'liczby' is not defined
>>>
```

Na początku wyświetlona została zawartość zmiennej. Po jej usunięciu interpreter wraca do stanu w jakim był w chwili jego uruchomienia! Próba wyświetlenia zmiennej liczby kończy się błędem.

Wszystkie zmienne jakie są w aktualnym stanie zdefiniowane w interpreterze można wyświetlić wbudowaną w niego funkcją dir().

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>>
```

Możemy sprawdzić kończąc działanie interpretera (Ctrl + D) i uruchamiając go na nowo, że powyższe identyfikatory są wbudowane (istnieją już na starcie).

2.Skrypty, programy wsadowe i pliki

Program wsadowy[36] nie komunikuje się z użytkownikiem w trakcie działania. Czyta dane z zadanego wejścia (zazwyczaj urządzenie zewnętrzne) i po przetworzeniu zapisuje je na zadanym wyjściu (zazwyczaj urządzenie zewnętrzne). Tak działały najstarsze programy, które przetwarzały dane zapisywane na taśmach magnetycznych (wcześniej – kartach perforowanych). Dane te nazwano plikami (mówi się też o plikach danych). Pojęcie pliku (ang. file) uogólniono na każdą kolekcję wielu danych, zapisanych w znany sposób i zazwyczaj oznaczonych nazwą. Sekwencyjny zapis i odczyt plików jest podstawowym sposobem operowania na dużych ilościach danych.

Spróbujmy stworzyć taki przykładowy program, który przetworzy nam listę osób na zbiór adresów mailowych. Załóżmy, że mamy listę w pliku csv (zob przykład 3.1 - mediana):

[36] https://pl.wikipedia.org/wiki/Program_wsadowy

Przykład 11.1a: **em1**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import csv

def doit_file(filename):
    csvfile=open(filename)
    fileout = open('email.txt','w')
    for row in csv.reader(csvfile, delimiter=';'):
        fileout.write('%s %s<%s>\n' % (row[0],row[1],row[2]))
    fileout.close()

if __name__ == "__main__":
    if len(sys.argv)>1:
        doit_file(sys.argv[1])
    else:
        print("Wywolanie z parametrem: nazwa pliku listy")
```

Wejście (zawartość pliku csv, którego nazwę podaliśmy jako parametr):

```
Jan;Kowalski;jk@gmail.com
Józef;Nowak;jn@gmail.com
```

Wyjście (zawartość pliku email.txt):

```
Jan Kowalski<jk@gmail.com>
Józef Nowak<jn@gmail.com>
```

Objaśnienie

1. Załóżmy, że program zapisaliśmy w pliku o nazwie „em1.py”. Jego wywołanie następuje poprzez podanie interpretera z nazwą pliku programu oraz nazwą pliku z danymi. Uruchom konsolę systemu operacyjnego (pod Windows poleceniem cmd) a następnie wydaj polecenie (zakładając, że dane są w pliku lista.csv, a program w pliku em.py):

```
python em.py lista.csv
```

1. Pierwszy wiersz zawiera nazwę interpretera. Jego wpisanie pozwala w systemach typu Unix (na przykład Linux) wywoływać program bez jawnego podawania interpretera. Na przykład:\ ∼/**em.py lista.csv**\ Przedrostek ∼/ oznacza tutaj, że plik jest w naszym katalogu domowym (w przeciwieństwie do Windows bieżący katalog nie jest domyślnym dla poszukiwania programu).\ Dodatkowo trzeba pamiętać, że plik programu musi mieć atrybuty[37] pozwalające na jego uruchomienie.
2. Jeśli zmienna __name__ (systemowa – wbudowana w Pythona) zawiera nazwę "__main__" - to oznacza, że moduł został uruchomiony jako główny moduł programu.
3. Zmienna sys.argv zawiera listę parametrów z jakimi został wywołany program. Zerowy element w liście zawiera nazwę uruchomionego programu. Gdy wywołano program:\ **./em.py lista.csv**\ to sys.argv[0] ma wartość './em.py' a sys.argv[1] ma wartość 'lista.csv'\ Wiersz poleceń może mieć więcej parametrów. Istnieje moduł **getopt**, ułatwiający ich analizę (zob.opis w podręczniku „Zanurkuj w Pythonie"[38]).
4. W funkcji doit_file otwieramy plik i kojarzymy z nim obiekt reader z modułu csv, który przetwarza przeczytane wiersze na listy.
5. Do pliku fileout zapisujemy tekst. Znaki \n oznaczają koniec wiersza.

3.Potoki i gniazda.

Teksty które program wypisuje wskutek użycia polecenia print są traktowane tak jak pliki (ciąg znaków tworzących plik tekstowy). Jest to standardowy plik wyjściowy dostępny pod zmienną stdout z modułu sys. Istnieje też standardowy plik wejściowy

sys.stdin. Używanie tych dwóch plików umożliwia korzystanie z mechanizmów systemu operacyjnego do przekierowania strumienia danych. Można na przykład skierować wyjście z programu do pliku używając operatora >.

```
program1 > plik1.txt
```

Efektem powyższej komendy będzie zapisanie wyniku działania programu (standardowej wyjście) **program1** do pliku **plik1.txt**.

Można także łączyć programy w ten sposób, że standardowe wyjście z jednego staje się standardowym wejściem drugiego. Zapisujemy to operatorem | (potok). Na przykład:

```
program1 | program2.
```

Zmodyfikujmy nieco poprzedni przykład. Aby zastosować standardowe wejście i wyjście.

[37] https://pl.wikipedia.org/wiki/Chmod
[38] https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie/Obs%C5%82uga_argument%C3%B3w_linii_polece%C5%84

Przykład 11.1b: **em2.py**

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import sys
import csv
def doit_std():
    for row in csv.reader(sys.stdin, delimiter=';'):
        print('%s %s<%s>' % (row[0],row[1],row[2]))
if __name__ == "__main__":
    doit_std()
```

Wyjście:

Sprawdźmy działanie programu komendą:

```
cat lista.csv | ./em2.py
```

dla Windows:

```
type lista.csv | python em2.py
```

Uzyskamy na ekranie wynik:

```
Jan Kowalski<jk@gmail.com>
Józef Nowak<jn@gmail.com>
```

Jeśli podamy komendę:

```
cat lista.csv | ./em2.py > maile.txt
```

to wynik zostanie wpisany do pliku „maile.txt”.

Podobnym do powyższego (łączenia przez pliki standardowe) mechanizmem komunikacji z programem są gniazda (socket). Zamiast uruchamiać program za każdym razem, gdy jest potrzebny – możemy zbudować program, który stale „rezyduje” w pamięci czekając na polecenia. Polecenia przesyła się przez gniazdo. Tą samą drogą otrzymuje się zazwyczaj wyniki. Gniazda w systemie operacyjnym mają swoje numery (podobnie jak pliki). Możemy je wykorzystywać do komunikacji między komputerami – odwołując się do nich poprzez numer i adres sieciowy komputera na którym gniazdo zostało utworzone.

Przykład:

http://www.python.rk.edu.pl/w/p/python-i-programowanie-sieciowe/

4.Obiekty

Programowaniu obiektowemu został poświęcony cały odrębny rozdział podręcznika. Tego typu programowanie jest w tej chwili powszechnie stosowane. Jednym z takich zastosowań jest tworzenie graficznych interfejsów do komunikacji z programem. Elementy interfejsu (takie jak okna, pola, przyciski, etykiety) są traktowane jak obiekty, których własności ustawiamy dla uzyskania odpowiedniego wyglądu. Natomiast metody obiektów służą do sterowania ich zachowaniem.

Aby ułatwić sobie tworzenie takich interfejsów, wykorzystuje się gotowe biblioteki programów. Taką popularną biblioteką jest wxPython[39].

A oto najprostszy przykład jej użycia.

Przykład 11.2a: **wx_obj**

```
1  #!/usr/bin/env python
2  import wx
3
4  # tworzymy obiekt aplikacji graficznej
5  app = wx.App(0)
6  # obiekt okna (klasa Frame)
7  frame = wx.Frame(None, -1, "Tytul okna")
8  frame.Show(True)
9
10 # panel = obiekt wnetrza okna
11 panel = wx.Panel(frame, -1)
12 # obiekt etykiety w oknie
13 label1 = wx.StaticText(panel, -1, "Etykieta")
14
15 # umieszczamy panel w oknie
16 frame.panel = panel
17
18 # okno w aplikacji
19 app.SetTopWindow(frame)
20 app.MainLoop()
```

Objaśnienie.

Aplikacja graficzna wykonuje pętlę (**MainLoop**) czekając na działania ze strony użytkownika. Poleceniem SetTopWindow wyświetlamy okno (**frame**) w którym ma zachodzić interakcja. Wnętrze okna nazywa się panelem i jest obiektem typu **wx.Panel**. Na panelu umieszcza się kontrolki (ang. **widgets**), które są odpowiedzialne za poszczególne elementy wizualne. Tu użyliśmy kontrolki **wx.StaticText** do umieszczenia na panelu etykiety.

[39] http://www.w3ii.com/pl/wxpython/default.html

Zauważmy, że biblioteka **wx** stosuje trzy rodzaje powiązania obiektów ze sobą. Może to następować poprzez parametr przy tworzeniu obiektu (jak w instrukcji **wx.Panel(frame, -1)** – podajemy w parametrze okno dla którego tworzymy panel). Gdy to powiązanie jest zmienne – jak w przypadku otwartego okna – stosujemy odrębną funkcję – jak w instrukcji app.SetTopWindow(frame). Trzecim sposobem jest bezpośrednia zmiana własności. Na przykład okno ma jeden panel dostępny przez własność o nazwie panel. Możemy tą własność ustawiać jak w instrukcji frame.panel = panel (zauważmy, że frame.panel i panel to dwie różne zmienne).

Powyższy przykład celowo został napisany w ten sposób, że ustawienie wszystkich elementów graficznych następuje na zewnątrz obiektu aplikacji graficznej. Jednak najczęściej stosuje się tworzenie obiektów pochodnych (poprzez dziedziczenie) i ustawianie elementów graficznych wewnątrz konstruktora (__init__) lub specjalnej metody **OnInit**. Zauważmy, że w ten sposób następuje izolowanie kodu dotyczącego poszczególnych klas i obiektów. Program staje się o wiele bardziej czytelny (co by było, gdybyśmy w poprzednim przykładzie inicjowali 50 okienek z 10 kontrolkami każde?).

Przykład 11.2b: **wx_inherit**

```
1  #!/usr/bin/env python
2
3  import wx
4
5  class TestApp(wx.App):
6
7  def OnInit(self):
8    frame = wx.Frame(None, -1, "Tytul okna")
9    frame.Show(True)
10   panel = wx.Panel(frame, -1)
11   label1 = wx.StaticText(panel, -1, "Etykieta")
12   frame.panel = panel
13   self.SetTopWindow(frame)
14   return True
15
16 app = TestApp(0)
17 app.MainLoop()
```

Objaśnienie

Jest to ten sam przykład, co 9.2a, ale wszystkie operacje dotyczące aplikacji graficznej zostały zaimplementowane w metodzie OnInit klasy pochodnej do wx.App. Należy zwrócić uwagę na fakt, że zamiast do zmiennej app, wewnątrz klasy odwołujemy się do parametru self – pod którym kryje się oczywiście odwołanie do obiektu aplikacji. W analogiczny sposób można tworzyć klasy pochodne do innych elementów graficznych.

5.Zdarzenia

Programowanie obiektowe jest czasem utożsamiane z programowaniem sterowanym zdarzeniami. Bierze się to stąd, że częścią obiektów takich jak w powyższych przykładach jest obsługa zdarzeń. Co będzie gdy klikniemy w oknie lub zmienimy jego rozmiar? Zostanie wywołana odpowiednia metoda obiektu wyświetlającego zawartość okna. Aby to zilustrować, dodajmy do naszego okna kontrolkę przycisku.

Przykład 11.3: **wx_app**

```
#!/usr/bin/env python
import wx

class TestApp(wx.App):
  def OnInit(self):
    frame = wx.Frame(None, -1, "Tytul okna")
    frame.Show(True)
    panel = wx.Panel(frame, -1)
    self.label1 = wx.StaticText(panel, -1, "Etykieta")

    button1 = wx.Button(panel, id=wx.ID_ANY,
          label="Kilknij", pos=(225, 5), size=(80, 25))
    button1.Bind(wx.EVT_BUTTON, self.onButton1Click)

    frame.panel = panel
    self.SetTopWindow(frame)
    return True

  def onButton1Click(self, event):
    self.label1.SetLabel("Button pressed!")

app = TestApp(0)
app.MainLoop()
```

Objaśnienie

Kontrolkę przycisku tworzymy wykorzystując klasę wx.Button. W parametrach podajemy między innymi (opcjonalnie) pozycję (pos) i rozmiar (size).

W implementacji obiektu aplikacji pojawiała się metoda onButton1Click, która jest odpowiedzialna za obsługę zdarzenia. Wiążemy tą metodę ze zdarzeniami dotyczącymi przycisku metodą **Bind**:

```
button1.Bind(wx.EVT_BUTTON, self.onButton1Click)
```

Wewnątrz funkcji onButton1Click zaimplementowana została zmiana etykiety label1. Aby ułatwić odwołanie do etykiety, została ona zainicjowana jako własność obiektu (użycie przedrostka self).

6.Wyjątki

Zdarzenia w programie nie muszą wiązać się wyłącznie z obsługą graficznego interfejsu użytkownika. Bardziej fundamentalne znaczenie ma występowanie sytuacji wyjątkowych – takich jak błędy w programie lub danych. Co się stanie na przykład wtedy, gdy w przetwarzanym pliku csv napotkamy źle sformatowany wiersz? Albo podamy błędną nazwę pliku? Takie sytuacje możemy próbować przewidzieć, dodając do programu warunki sprawdzające poprawność. Ale jeśli w programie zrobimy błąd - wtedy trudno nawet oczekiwać, że taka sytuacja zostanie przewidziana. Można jednak przewidzieć pojawienie się jakiegoś (jakiegokolwiek) błędu i oprogramować obsługę takiej sytuacji.

Służą do tego wyjątki, które zostały szerzej opisane w poprzednim rozdziale. Wracamy do tego zagadnienia przede wszystkim dlatego, żeby pokazać nietypowy przepływ sterowania w przypadku wydarzenia się (lub zgłoszenia) wyjątku.

Poniżej uproszony nieco przykład z rozdziału poświęconego wyjątkom. Drukowanie kolejnych liczb pokazuje jak przebiega sterowanie.

Przykład 11.4: **exc11**

```
# -*- coding: utf-8 -*-
class BlednyKodPocztowy(Exception):

    def __init__(self, msg):
        Exception.__init__(self)
        print(5)
        self.msg = msg

class KodPocztowy():
    wartosc = '00-000'

    def set(self,kod):
        print(3)
        if (len(kod)<>6) or (kod[2]<>'-'):
            print(4)
            raise BlednyKodPocztowy('Poprawny format kodu: 99-999')
        print('Tu tylko gdy poprawny kod')
        self.wartosc=kod
```

```
19
20 kod=KodPocztowy()
21 print(1)
22 try:
23     print(2)
24     kod.set('22-22')
25     print('Tu tylko gdy poprawny kod')
26 except BlednyKodPocztowy as e:
27     print(e.msg)
28     print(6)
29 print(7)
```

Wyjście:

```
1
2
3
4
5
Poprawny format kodu: 99-999
6
7
```

Objaśnienie

Uzyskany wynik potwierdza, że w programie prawidłowo przewidziano przepływ sterowania oraz miejsca ominięte w razie błędu danych.

7.Systemy wielowarstwowe i rozproszone, interfejsy.

W podrozdziale dotyczącym gniazd wspomniano o komunikacji między programami w sieci. Ponieważ w typowym przypadku programujemy komunikację z programem napisanym przez kogoś innego, dąży się do ustalenia jak najbardziej uniwersalnych zasad komunikacji, zwanych interfejsami. Każdy z nas korzystając z internetu styka się z interfejsem między przeglądarką a serwerem sieciowym (najczęściej jest tu wykorzystywany protokół http lub https). Najczęściej jednak mamy do czynienia z programami odwołującymi się do baz danych. Do serwera na którym umieszczono bazę danych możemy odwoływać się przesyłając polecenia przez gniazda (cob wyżej). Łatwiej jednak używać bibliotek, które całą komunikację wykonają za nas, przekazując nam gotowe dane w formacie struktur Pythona (zob. rozdział poświęcony przetwarzaniu danych).

Innym przykładem rozproszonej aplikacji może być serwer stron internetowych. Elementy takiej aplikacji mogą rezydować na komputerach nawet na różnych kontynentach!

[v3]

Styl programowania

Należy pisać kod programów w taki sposób, by nadawał się do ponownego wykorzystania. Stąd reguła: *program piszemy raz, ale czytany może być wiele razy*. Musi zatem być on napisany zgodnie ze standardami, z użyciem zrozumiałych identyfikatorów i podzielony na fragmenty w sposób intuicyjnie prosty.

Stykają się tutaj dwie przeciwstawne tendencje. Z jednej strony przyjmuje się standardy prowadzące do ujednolicenia (nieważne w jakim języku programujesz – zasady są takie same). Z drugiej – wybieramy język Python między innymi z uwagi na wygodne mechanizmy. Pisząc program w Pythonie używamy zatem pewnych rozwiązań sprawiających, że kod ma swoją specyfikę. Jest „pythonowaty" (ang. pythonic). Zarówno te ogólne zalecenia jak i „pythonizmy[40]" składają się na coś, co można nazwać stylem programowania.

Niniejszy rozdział poświęcony jest temu zagadnieniu. Zawiera on nie tylko zbiór praktycznych zaleceń. Poznamy przy okazji kolejne – dotąd nie opisywane w tym podręczniku konstrukcje języka Python. Niektóre z tych konstrukcji (lub sposobów ich użycia) nie są zalecane. Opiszemy takie przypadki, aby z jednej strony było wiadomo czego konkretnie unikać, a z drugiej – aby pomóc w zrozumieniu kodu, w którym jednak takich konstrukcji użyto.

Na wstępie jedna ważna uwaga:

Opanowanie konwencji nie warunkuje umiejętności tworzenia programów. To jest proces, który zachodzi równolegle z rozwojem umiejętności programowania. Nie poprzedza i nie warunkuje tej umiejętności. To tylko dobre rady: jak postępować, aby nasz kod był coraz lepszy.

PEP8

Niniejszy podręcznik nie zawiera kompletnego opisu zalecanego stylu programowania. Pominięto między innymi konwencje związane z nazewnictwem zmiennych, stosowaniem odstępów etc…

Tego typu zalecenia można znaleźć w specyfikacji PEP8[41] (polskie omówienie jest na przykład na blogu blog.grzeszy.net[42] lub w podręczniku „Efektywny python …[43]"). PEP to skrót od „Python Enhancement Proposals" (propozycje rozwoju Pythona). Numerowane dokumenty PEP są gromadzone na stronie www.python.org/dev/peps/[44]. Ten z numerem 8 (PEP 8[45]) zawiera powszechnie

[40] http://python101.readthedocs.io/pl/latest/podstawy/przyklady/index5.html
[41] http://pep8.org/
[42] http://blog.grzeszy.net/moja-sciaga-z-pep-8-czesc-1
[43] http://helion.pl/ksiazki/efektywny-python-59-sposobow-na-lepszy-kod-brett-slatkin,efepyt.htm
[44] http://www.python.org/dev/peps/
[45] https://www.python.org/dev/peps/pep-0008

akceptowane zalecenia dotyczące stylu programowania . Na stronie pep8.org[46] jest promowany program pep8[47], który sprawdza nasz kod i drukuje zalecenia zgodne z pep8.

Jak go przetestować?

```
$ pip install pep8
$ pep8 mojprogram.py
Istnieje też program autopep8 który potrafi odpowiednio sformatować nasz kod:
$ pip install autopep8
$ autopep8 --in-place mojprogram.py
```

Zen Pythona

Kod programu w Pythonie wyróżnia się przejrzystością. To nie jest przypadek, ale efekt starań twórców tego języka i przyjętych przez nich założeń. Założenia te – w postaci szeregu aforyzmów zostały udostępniane jako Zen Pythona (nazwa pochodzi od buddyjskiej filozofii zen[48]). Aby je wyświetlić, należy napisać w wierszu poleceń interpretera Pythona:

```
>> import this
```

Wyświetli się spis zaleceń w języku angielskim:

> The Zen of Python, by Tim Peters
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one– and preferably only one –obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.

[46] http://pep8.org/
[47] https://github.com/jcrocholl/pep8
[48] https://pl.wikipedia.org/wiki/Zen

> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!

Tłumaczenie (http://blog.zabiello.com/2008/09/01/zen-of-python) na język polski:

```
Piękny jest lepszy niż brzydki.
Jawny jest lepszy niż domyślny.
Prosty jest lepszy niż złożony.
Złożony jest lepszy niż skomplikowany.
Płaski jest lepszy niż zagnieżdżony.
Rzadki jest lepszy niż zagęszczony.
Czytelność ma znaczenie.
Sytuacje wyjątkowe nie są na tyle wyjątkowe, aby łamać reguły.
Aczkolwiek pragmatyzm wygrywa z puryzmem sztywnego trzymania się reguł.
Błędy zawsze powinny być sygnalizowane.
No chyba, że zostaną celowo ukryte.
W obliczu dwuznaczności odrzuć pokusę aby zgadywać.
Powinien istnieć jeden, i najlepiej tylko jeden, oczywisty sposób do zrobienia d\
anej rzeczy.
Chociaż sposób ten nie musi być początkowo oczywisty jeśli nie jesteś Holendrem.
Teraz jest lepsze niż nigdy.
Chociaż nigdy jest często lepsze niż właśnie teraz.
Źle, jeśli implementację jest trudno wyjaśnić.
Dobrze, jeśli implementację jest łatwo wyjaśnić.
Przestrzenie nazw są świetnym pomysłem – stwórzmy ich więcej!
```

Nie ma potrzeby, aby uczyć się tych reguł. Jednak warto od czasu do czasu je sobie wyświetlić i zastanowić się, czy je stosujemy. W całym rozdziale będziemy o tych regułach pamiętać i czasem nawiązywać do nich. Na początek warto zwrócić uwagę na jedno: programowanie to nie tylko intelektualny wysiłek, ale też fajna zabawa. Dobry programista to pasjonat, który lubi to co robi. Ta pasja jest napędzana satysfakcją jaką daje dobrze działający program.

Lukier składniowy

Określenie „lukier składniowy[49]" (ang. syntactic sugar[50]) jest używane w odniesieniu do elementów składni języka, które są nadmiarowe, ale zostały wprowadzone dla wygody programisty. Czasem

[49] https://pl.wikipedia.org/wiki/Lukier_sk%C5%82adniowy
[50] https://en.wikipedia.org/wiki/Syntactic_sugar

może to być postrzegane jako zbędne udziwnienia. Jednak w Pythonie tego rodzaju rozszerzenia wprowadzane są zgodnie z powyższymi zasadami „zen" - więc zapewniają nie tylko większą wygodę, ale i wzrost czytelności. Dobrym przykładem takiego rozszerzenia są dekoratory[51], które poznaliśmy już wcześniej.

Określenie „lukier składniowy" nie jest precyzyjne – bo ma charakter publicystyczny a nie techniczny. Do tego typu rozwiązań można zaliczyć kilka konstrukcji o których będzie mowa w niniejszym rozdziale (wyrażenia „for" oraz „if", funkcje lambda). Z całą pewnością można tak nazwać poniższe dwa rozwiązania: podział tekstu na wiersze oraz łączenie porównań.

Dzielenie tekstu na wiele wierszy

Jednym z zaleceń PIP8 jest ograniczenie długości wierszy do 80 znaków. Co jednak zrobić, gdy ten wiersz jest bardzo długi? Pokazuje to poniższy przykład:

Przykład 12.1: **break_line**

```
1  # uzycie znaku kontynuacji \ w instrukcji
2  k1=1,2,3,\
3   4,5
4  print(k1)
5  #nie dokonczona instrukcja
6  k2=('a','b','c'
7  ,'d')
8  print(k2)
9  # lukier skladniowy
10 dlugi_napis = ("to jest bardzo długi "
11 "tekst")
12 print(dlugi_napis)
13 dlugi_napis = ("alternatywa: bardzo długi \
14   tekst")
15 print(dlugi_napis)
```

Wyniki:

```
(1, 2, 3, 4, 5)
('a', 'b', 'c', 'd')
to jest bardzo długi tekst
alternatywa: bardzo długi     tekst
```

Objaśnienie:

[51] http://www.rwdev.eu/articles/decorators

1. W przypadku instrukcji sprawa jest prosta. Wiersz można podzielić na kilka krótszych używając znaku kontynuacji \ (krotka k1).
2. Znak kontynuacji nie jest potrzebny gdy interpreter „wie", że instrukcja nie jest zakończona i bedzie kontynuowana (krotka k2 nie kończy sie nazwiasem zamykającym).
3. W przypadku dłuższy łańcuch znaków (stałych tekstowych) można stosować także backslash (\) albo łączenie tekstów (metodą join). Ale to wprowadzają dodatkowe komplikacje. Na przykład jeśli nie dosuniemy nowego wiersza do lewego marginesu - powstaną spacje. Istnieje prostsze rozwiązanie: kilka łańcuchów znaków ujętych w nawiasy okrągłe jest traktowane jak jeden łańcuch (dlugi_napis). Należy zwrócić uwagę na brak przecinka rozdzielającego łańcuchy znaków. Tym różni się zapis w wielu wierszach jednej stałej od krotki zawierającej wiele stałych.

Łączenie porównań

Jeśli chcemy sprawdzić, czy zawartość zmiennej mieści się w określonym zakresie, można użyć dwóch porównań złączonych spójnikiem **and**:

Przykład 12.2: **ex_cond**

```
1  wiek=20
2  if 18 < wiek and wiek <= 25:
3      print('OK (1)!')
4
5  # to samo ale krocej:
6  if 18 < wiek <= 25:
7      print('OK (2)!')
```

Objaśnienie:

Python połączenie takich porównań w łańcuch. Tego nie ma (chyba?) w żadnym innym języku programowania. Dlatego budzi to kontrowersje. Niemniej warto wiedzieć, że coś takiego istnieje.

Generowanie wielu powtórzeń tego samego

Operator mnożenia (*) w Pythonie może być używany do list:

Przykład 12.3: **n_list**

```
1  lista10 = [0] * 10
2  print(lista10)
```

Wynik:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Rozpakowywanie sekwencji

Rozpakowywanie krotek to często spotykane rozwiązanie, na które czasem nawet nie zwracamy uwagi z uwagi na jego naturalność. Korzystamy z tego przy przeglądaniu sekwencji krotek (na przykład czytanych z bazy danych), czy zamianie wartości zmiennych.

Przykład 12.4: **upack_tuple**

```
krotki=((1,11),(2,12),(3,13))
aa=bb=0
for a,b in krotki:
    print('a=%s; b=%s' % (a,b))
    aa,bb=aa+a,bb+b
print(aa,bb)
aa,bb=bb,aa
print(aa,bb)
```

Wynik:

```
a=1; b=11
a=2; b=12
a=3; b=13
(6, 36)
(36, 6)
```

Objaśnienie:

1. Pętla for działa w przypadku sekwencji krotek podobnie jak przy odczycie z bazy danych. Podając identyfikatory poszczególnych elementów dokonujemy ich rozpakowania. Każdy element staje się zmienną o podanym identyfikatorze.
2. Rozpakowywania krotek można użyć na przykład do zamiany wartości dwóch zmiennych. Lewa strona równania w przedostatnim wierszu zawiera rozpakowane kroki (wartości w zmiennych). Po prawej stronie równania mamy nową krotkę (a nie tą rozpakowaną). Ale oczywiście można wykorzystać zmienne z rozpakowania - to umożliwia zamianę wartości.

Można także rozpakowywać listy (operatorem *) i słowniki (operator **). Ale to jest rzadko używane i w zasadzie tylko do przekazywania parametrów:

Przykład 12.5: **upack_list**

```
1  def abc(a,b,c=0):
2    print(a,b,c)
3
4  def upack():
5    lista = [1,3,5]
6    abc(*lista)
7    slownik={'b':4,'a':5}
8    abc(**slownik)
9
10 upack()
```

Wynik:

```
(1, 3, 5)
(5, 4, 0)
```

Idiomy w programowaniu

Idiomy[52] to pewne „osobliwości języka" pozwalające w sposób nietypowy coś wyrazić. Programiści Pythona nazywają tak („Python Idioms[53]") ideę wybierania spośród różnych możliwości zapisu (wzorców) takiego, które jest optymalny (zalecany). Poniższa lista idiomów nie jest kompletna (to nie jest formalny standard).

Logiczna wartość różnych danych

Dane typu logicznego (boolean) mogą być pamiętane w zmiennych tak samo jak dane innych typów. Można więc zapisać:

```
1  x=int(raw_input('liczba większa od 10='))
2  dobra=(x>10)
3  if not dobra:
4     print('Liczba niepoprawna')
```

Warunek w instrukcji if można zapisać także tak: if not (dobra == True), lub tak: if dobra == False. Jednak zaleca się korzystanie z tej pierwszej możliwości.

[52] https://pl.wikipedia.org/wiki/Idiom

[53] http://highenergy.phys.ttu.edu/~igv/ComputationalPhysics/Tutorials/Tutorial2/CommonPythonIdioms.html

Zaleca się przy sprawdzaniu warunków wykorzystywanie wartości logicznej zmiennej, a nie wyrażenia logicznego.

Nie tylko zmienne zawierające wartość True albo False posiadają wartość logiczną. Na przykład wartość pusta (None) lub pusta lista jest traktowana na równi z fałszem.

Zamiast:

```
1  if  len(lista) != 0:
2      print('niepusta')
```

można (i warto) zapisać:

```
1  lista=[1,]
2  if lista:
3      print('niepusta')
```

Poniższa tabelka zawiera spis wartości logicznej True (prawda) lub False (fałsz) różnych wyrażeń:

True	False
Niepusty łańcuch znaków	Pusty łańcuch znaków
Liczba różna od zera	Liczba 0 (lub 0.0)
Kolekcja (lista, słownik etc.) niepusta:	Kolekcja pusta (`[]` , `()` , `{}` , `set()`):
len(x) > 0	len(x) == 0
-	None

Przykład 12.6: bval

```
1  imie = 'Jan'
2  zwierz = ['Pies', 'Kot']
3  posiadanie = {'Jan': 'Kot', 'Adam': 'Pies'}
4
5  # Zapis zalecany
6  if imie and zwierz and posiadanie:
7    print('Ktoś ma zwierzęta!')
8
9  # Zapis równoważny (jednak nie zalecany)
10 if imie != '' and len(zwierz) > 0 and posiadanie != {}:
11   print('Ktoś ma zwierzęta!')
```

Można też sprawdzać wartość logiczną obiektów! W chwili porównania wywoływana jest metoda

wewnętrzna (systemowa[54]) __bool__ (Python 3.x) lub __nonzero__ (Python 2.x). Można oczywiście zdefiniować swoją implementację tych metod.

Jeśli obiekt służy nam do zapamiętywania danych (pojemnik / container) – można sprawdzać ilość tych danych. Służy do tego metoda __len__:

Przykład 12.7: **o_bool**

```
class Test(object):
# to zadziała zarówno z Python2x jak i Python 3#
    def __init__(self, value):
        self.value = value

    def __bool__(self):
        return False

    def __nonzero__(self):
        return self.__bool__()

class Queue(object):
    def __init__(self, iterable):
        self.queue = iterable
    def __len__(self):
        return len(self.queue)

print(bool(Test(0)))
print(bool(Queue([])))
print(bool(Queue([1,3,2])))
```

Wynik:

```
False
False
True
```

Objaśnienie:

Obiekt klasy Test zawsze zwraca False. Ustala tą wartość metoda __bool__ lub __nonzero__. Natomiast wartość obiektu klasy Queue zależy od tego, czy zainicjowano go z pustą listą.

Co można zapisać w jednym wierszu? Wyrażenia if.

Python pozwala na umieszczenie kilku wierszy kodu w jednym z użyciem średników. Zamiast więc pisać:

[54] https://docs.python.org/3/reference/datamodel.html

```
a=3
b=raw_input('parametr b=')
print(a*int(b))
```

możemy napisać:

```
a=3;b=raw_input('parametr b=');print(a*int(b))
```

Można też w jednym wierszu zawrzeć instrukcję wraz z jednowierszowym blokiem. Wtedy nie używamy średnika:

```
if mianownik==0: print('Nie dziel przez zero')
```

Kiedy należy stosować taki zapis? W zasadzie nigdy – **on nie jest zalecany**.

Pamiętaj (zen): „*rzadki jest lepszy niż zagęszczony*" (co należy rozumieć jako pochwałę przejrzystości). Czyli twój kod powinien mieć dużo wolnej przestrzeni. Nie staraj się upchnąć (zagęścić) jak najwięcej znaków w jednym wierszu. Stosuj też przerwy (spacje), a jeśli dodanie wolnego wiersza zwiększa czytelność – nie wahaj się go używać.

Jednak możemy czasem (jeśli nie wpływa to na czytelność kodu) zamienić kilka wierszy instrukcji na jeden. Weźmy na przykład fragment kodu, który zwraca nam łaciński skrót (stosowany najczęściej tam gdzie nie używa się zegara 24 godzinnego – na przykład w USA) oznaczający przed południem (PM) i po południu (AM):

```
from datetime import datetime
now = datetime.now()
if (now.hour>=12):
  pm_am = 'PM'
else:
  pm_am = 'AM'
print(pm_am)
```

Możemy zamiast instrukcji if użyć wyrażenia if. Jest to konstrukcja języka Python pozwalająca w miejsce wartości wpisać sposób jej wyliczenia z użyciem instrukcji takich jak if lub for.

Przykład 12.8: **ex_if**

```
from datetime import datetime
now = datetime.now()
pm_am = 'PM'  if (now.hour>=12) else  'AM'
print(pm_am)
```

Objaśnienie

Najpierw podajemy wartość, potem warunek zapewniający, że ta wartość będzie użyta, a następnie – alternatywę. Zwróć uwagę na to, że nie używa się w tym wypadku dwukropka po słowach kluczowych **if** i **else**.

Taką konstrukcję można użyć na przykład do odmiennego obsłużenia wartości pustej. Na przykład:

```
wynik += '' if symbol is None else symbol
```

W powyższej instrukcji warto zwrócić uwagę na zapis „symbol is None". Jest on równoważny z „symbol == None", ale bardziej czytelny. Symbol pusty (None) ma swoją interpretację w pamięci komputera, ale stosując operator „is" (jest) lub „is not" (nie jest) abstrahujemy od tego. Ważne jest także to, że taka konwencja (is / is not) jest stosowana w bazach danych (SQL), gdzie nie można stosować zwykłych porównań z symbolem pustym (rolę „None" pełni tam „null").

Konstrukcji wyrażenia **if** nie można komplikować (tak jak w zwykłej instrukcji warunkowej). Chodzi wyłącznie o wybór z dwóch wartości. Nie możemy nawet łączyć wyrażenia if z wyjątkami. Taki zapis **NIE JEST POPRAWNY**:

```
wynik_dzielenia = licznik / mianownik if mianownik != 0 else raise ZeroDivisionE\
rror
```

Wyrażenia for

Podobnym do wyrażenia if jest tworzone na bazie pętli wyrażenie **for**. Stosuje się je do generowania list. Podobnie jak w wyrażeniu if najpierw podajemy wyrażenie wyliczające wartości, a później iterację (for) pozwalającą wygenerować wiele tych wartości.

Na przykład listę kwadratów liczb od 0 do 9 można zapisać:

Przykład 12.9: **ex_for**

```
print([n ** 2 for n in range(10)])
# Możemy to łączyć z wyrażeniami if.
# Na przykład kwadraty liczb nieparzystych:
print([n ** 2 for n in range(10) if n % 2])
```

wynik:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 9, 25, 49, 81]
```

Objaśnienie:

Utworzono dwie listy używając wyrażeń for.

Wyrażenia for stosuje się do generowania list. Jeśli wyrażenie jest zbyt złożone - lepiej użyć zwykłej pętli lub zdefiniować funkcję generującą.

Aby zilustrować problem komplikacji załóżmy, że chcemy wygenerować sześć przypadkowych liczb wskazujących kierunek ruchów pionka po szachownicy warcabów (-1 = w lewo, +1 = w prawo).

Możemy to zapisać:

```
import random
kierunek=[ random.choice([-1,1]) for i in range(6) ]
```

Funkcja choice z modułu random dokonuje przypadkowego wyboru z listy podanej jako parametr (tutaj: [-1,1]). Ale co zrobić, aby uniknąć wyjścia poza szachownicę (x<1 lub x>8)? Możemy zapisać to następująco:

```
# 6 przypadkowych ruchow pionka na szachownicy warcabów
# startujemy z pola o wspolrzednych (2,2)
# Nieprzejrzysty kod!!!!
import random
x=y=2
wspolrzedne=[]
for i in range(6):
  x += (-1 if x>1 else 1) if random.choice([-1,1])==-1 \
     else (1 if x<8 else -1)
  y+=1
  wspolrzedne.append((x,y))
```

Tylko czy to jest na pewno uproszczenie? Użyta w poprzednim przykładzie instrukcja zwiększania wartości x jest poprawna, ale zbyt złożona. A przecież można zapis jeszcze bardziej skomplikować:

```
# Zły kod!!!!
x=2
def nx(d):
  global x
  x+=d
  return x

wspolrzedne = [ (nx((-1 if x>1 else 1) if random.choice([-1,1])==-1 else (1 if x\
<8 else -1)),y) for y in range(2,8)]
```

Mamy kod krótki ale nieprzejrzysty. Zdecydowanie bardziej przejrzyste jest tutaj zastosowanie instrukcji for, a nie wyrażenia for:

Przykład 12.10: **warcaby1**

```
1  # 6 przypadkowych ruchów pionka na szachownicy warcabów
2  #
3  import random
4  kierunek=[ random.choice([-1,1]) for i in range(6) ]
5  wspolrzedne=[]
6  x=2
7  for y in range(6):
8    if (x + kierunek[y]) == 0:
9      x = 2
10   elif (x + kierunek[y] > 8):
11     x = 7
12   else:
13     x += kierunek[y]
14   wspolrzedne.append([x,y+2])
```

Kod dłuższy ale przejrzysty (bardziej „rzadki”).

Anonimowe funkcje lambda

W jednym wierszu można także zdefiniować funkcję! Taka anonimowa (pozbawiona nazwy) funkcja często występuje w połączeniu z wyrażeniem for.

Przykład 12.11: **lambda**

```
1  [(lambda x: x**3)(n) for n in range(10)]
```

wynik:

```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Objaśnienie:

W jednym wierszu zdefiniowano funkcję (lambda) i użyto ją do wygenerowania wartości elementów listy.

Nie jest to zbyt przejrzyste. Można to uprościć zapamiętać funkcję w zmiennej (kojarząc ją w ten sposób z nazwą).

```
cube = lambda x: x**3
[cube(n) for n in range(10)]
```

Tylko czy nie prościej po prostu zdefiniować zwykłą funkcję?

```
# Zamiast: cube = lambda x: x**3
# użyj:
def cube(x):
    return x**3
[cube(n) for n in range(10)]
```

Zalecenie: **unikaj stosowania funkcji anonimowych lambda!**

Uproszczenie instrukcji warunkowych

Jak już wspomniano (przy okazji opisu wartości logicznych różnych danych) – zaleca się unikać operatorów porównania tam, gdzie są zbędne.

Czasem w celu osiągnięcia bardziej przejrzystego kodu można nawet podzielić wyrażenie na kilka – wprowadzając zmienne do zapamiętania wyniku częściowego.

Zamiast

```
if <złożony warunek> and <inny złożony warunek>:
    # blok instrukcji
```

Zapisz:

```
warunek1 = <złożony warunek>
warunek2 = <inny złożony warunek>
if warunek1 and warunek2:
    # blok instrukcji
```

Na przykład algorytm rozrzucania pionków od warcabów na planszy:

Przykład 12.12: **warcaby2**

```
1   import random
2   #  plansza = [[0 for _ in range(8)] for _ in range(8)]
3   plansza = [8 * [0] for _ in range(8)]
4   ilosc = 0
5   while (ilosc < 12):
6     x = random.choice(range(8))
7     y = random.choice(range(8))
8   #    if (((x+y) % 2) == 1) and (plansza[y][x] == 0):
9     czarne = ((x+y) % 2) == 1
10    if (czarne) and (plansza[y][x] == 0):
11      plansza[y][x] = 1
12      ilosc += 1
13  for y in range(8):
14    print(plansza[y])
```

Objaśnienie

Pierwsza część warunku (zob. wiersz komentarza) sprawdza, czy pole jest czarne. Użycie zmiennej nie tylko upraszcza warunek, ale pozwala zachować informację o polu i stanowi dodatkowe objaśnienie (nazwa zmiennej).

Eliminacja zbędnych pętli przy pomocy funkcji wbudowanych

W Pythonie mamy kilka funkcji wbudowanych[55], działających na kolekcjach:

- any() - jeden z elementów sekwencji jest równoważny True
- all() - wszystkie elementy sekwencji
- min() - najmniejszy element
- max() - największy element
- sum() - suma elementów

Ich używanie pozwala uniknąć zbędnych pętli.

[55] https://docs.python.org/2/library/functions.html

Przykład 12.13: **reduct_for**

```
data=[2,5,32,11,6]
num_elements=len(data)
# Zbędna pętla – wyliczenie śrenie:
total = 0
for n in data:
  total += n
mean = total / len(data)

# Lepsze rozwiązanie:
mean1 = sum(data) / len(data)
print(mean,mean1)

# Pętla dla fragmentu listy – średnia pierwszych N elementów
num_elements=3
total = 0
for i in range(num_elements):
  total += data[i]
mean = total / num_elements
# równoważne rozwiązanie
mean1 = sum(data[:num_elements]) / num_elements
print(mean,mean1)
```

Używaj jeśli to możliwe funkcji agregujących zamiast przeglądać kolekcję danych w pętli.

Korzystaj z wyjątków!

Często w programach sytuacja wyjątkowa jest generowana próbą użycia nie istniejącego obiektu (na przykład element listy o indeksie poza zakresem). Zaleca się zatem (ale nie w Pythonie), aby sprawdzać przed użyciem (**LBYL** = “Look Before You Leap”). Alternatywą jst zasada: Prościej spróbować niż sprawdzać (**EAFP** = “It’s Easier to Ask for Forgiveness than Permission”).

Przykład 12.14: **value_except**

```
# dobrze
d = {'x': '5'}
try:
  value = int(d['x'])
except (KeyError, TypeError, ValueError):
  value = None
print(value)
# gorzej
d = {'x': '5'}
if 'x' in d and isinstance(d['x'], str) and d['x'].isdigit():
  value = int(d['x'])
else:
  value = None
```

Oczywiście można to połączyć z blokami else (inny wyjątek) i finally (wykonywanymi na koniec instrukcji):

Przykład 12.14a: **value_except2**

```
d = {'x': '5'}
try:
  value = int(d['x'])
except (KeyError, TypeError, ValueError):
  value = None
else:
 value=-1
finally:
  print(value)
```

W Pythonie obsługa wyjątków nie jest kosztowna, dlatego lepiej założyć poprawność danych niż sprawdzać to przed użyciem. Jednak musimy pamiętać o uzyciu wyjątków - żeby zareagować w razie niepoprawnych danych.

Jak importować

Chcąc zaimportować większą ilość identyfikatorów z jakiegoś modułu możemy teoretycznie wykonać:

```
from module import *
```

To nie jest zalecane, ale zazwyczaj działa. Gwiazdka oznacza wszystko co jest do zaimportowania. Czyli nie wiemy co konkretnie. Dochodzi do bałaganu (zaśmiecenia przestrzeni nazw). Czasem może spowodować zmianę znaczenia identyfikatorów, jakiej się nie spodziewamy.

Jeśli chcesz zaimportować wiele nazw, to już lepiej importować cały moduł. Wewnętrzne obiekty będą dostępne jeśli poprzedzimy je nazwą modułu:

```
import module
module.name
```

Jeśli nazwa modułu jest długa, możemy zastosować alias (pseudonim):

```
import module_with_very_long_name as mod1
mod1.f()  # uzycie pseudonimu mod1 i wywołanie funkcji
```

Nie używaj gwiazdki do zaimportowania wszystkiego. Możesz natomiast zaimportować sam moduł, używając go przedrostka do udostępnianych funkcji. Przy dłuższych nazwach modułów używaj aliasów.

Programowanie obiektowe

Prawdziwie wielkie możliwości i swobodę w komplikowaniu lub upraszczaniu kodu daje nam jednak dopiero zastosowanie klas i obiektów. Ta swoboda jest na tyle duża, że stworzono cały szereg zaleceń ograniczających jej nadużywanie. Na szczęście Python zen Pythona sprawia, że część z tych zasad nie ma większego znaczenia i nie musimy sobie nimi zaprzątać głowy.

Na początek zobaczmy jak można obiektowo rozwiązać opisane wcześniej zadanie generowania ruchów pionka (dbając równocześnie, aby pionek nie wypadł poza planszę).

Przykład 12.15: **warcaby3**

```
import random
PRAWO=1
LEWO=-1

class Pionek:
  x=y=1
  def __init__(self,x,y):
     self.x=x
     self.y=y

  def losowe_ruchy(self):
    while self.y<8:
      self.y +=1
      dx=random.choice([PRAWO, LEWO])
      if (self.x+dx==0) or (self.x+dx>8):
        self.x -= dx
      else:
        self.x += dx
      yield (self.x,self.y)

p = Pionek(1,1)
wspolrzedne = [ _ for _ in p.losowe_ruchy ()]
```

Klasę Pionek można umieścić w odrębnym module i zaimportować go w dowolne miejsce w jakim będzie używana. Czy jednak projektując taką klasę nie warto by umieścić w niej innych atrybutów do wykorzystania w przyszłości? Na przykład czy pionek jest zbity, albo jego kolor? Nie!

Pamiętajmy: prosty jest lepszy niż złożony. Jeśli komuś będą potrzebne dodatkowe atrybuty, to stworzy klasę pochodną, albo doda wymaganą funkcjonalność do istniejącej klasy.

Ta zasada postępowania została nazwana **YAGNI**[56] (skrót od „You ain't gonna need it"). Czyli: **nie pisz kodu który aktualnie nie jest wymagany – bo może się przydać**. Napiszesz gdy będzie potrzebny – wtedy na pewno będzie lepiej pasował do tych potrzeb. Zastosowanie ma tu też reguła **KISS**[57] (od ang. Keep It Simple, Stupid), czyli „nie komplikuj, głuptasku".

[56] http://www.foundbit.com/pl/zasoby/programowanie/podstawy/articles/zasady-kodowania.html
[57] https://pl.wikipedia.org/wiki/KISS_%28regu%C5%82a%29

SOLID i abstrakcje

Zalecenia przejrzystego programowania obiektowego (nie tylko w Pythonie) określone zostały akronimem SOLID. Nazwa ta kryje 5 zasad (przypominamy: zasady a nie bezwzględnie obowiązujące reguły!):

- SRP - Single responsibility principle (zasada jednej odpowiedzialności[58])
- OCP - Open/closed principle (otwarty na rozszerzenia ale zamknięty na zmiany[59])
- LSP - Liskov substitution principle (zasada podstawienia Liskov[60])
- ISP - Interface segregation principle (zasada segregacji interfejsów[61])
- DIP - Dependency inversion principle (zasada odwrócenia zależności[62])

SRP

Weźmy banalny przykład:

```
class Pionek:
  def ustaw(self,kolor=None,pozycja=(1,1)):
     if Kolor: self.kolor=kolor
     (self.x,self.y)=pozycja
```

Czy za każdym razem gdy chcemy przesunąć pionek, musimy ustawiać kolor? Unikniemy tego rozdzielając funkcje tak, aby każda miała jedną odpowiedzialność:

```
class Pionek:
  def ustaw_kolor(self,kolor):
     self.kolor=kolor

  def ustaw_pozycje(pozycja):
     (self.x,self.y)=pozycja
```

OCP

Obiekty powinny być tak napisane, aby łatwo było je rozszerzać bez konieczności grzebania w kodzie. Preferuje się zatem dziedziczenie obiektów i dekoratory. W Pythone dotyczy to głównie używania pakietów bibliotecznych. Rozważmy dwie strategie:

- zainstaluj, zaimportuj, stwórz obiekt pochodny

[58] https://pl.wikipedia.org/wiki/Zasada_jednej_odpowiedzialności
[59] https://pl.wikipedia.org/wiki/Zasada_otwarte-zamknięte
[60] https://pl.wikipedia.org/wiki/Zasada_podstawienia_Liskov
[61] http://www.dworld.pl/blogEntry/jump/8716?nomenu=true
[62] http://www.dworld.pl/blogEntry/jump/8716?nomenu=true

- skopiuj, zmodyfikuj, użyj

Zdecydowanie wybieramy pierwszą z nich. To tylko przykład zastosowania zasady OCP. W dużych projektach opartych na pracy zespołowej może mieć ona szersze zastosowanie.

Aby zilustrować tą zasadę zastosujmy drugą z powyższych strategii (dziedziczenie) dla prostego obiektu pionka.

Przykład 12.16: **warcaby4**

```
class AbsPionek:
  x=y=1

  def __init__(self,x,y):
     self.x=x
     self.y=y

  def przesun(self):
    raise NotImplementedError()

  def losowe_ruchy(self):
    return []


#Aby zaimplementować ruch dla pionka warcabów stosujemy dziedziczenie:

class PionekWarcabow(AbsPionek):

  def przesun(self,kierunek):
  # kierunek =-1 (lewo) lub +1 (prawo)
    if self.y<8:
      self.y +=1
     self.x += kierunek
```

LSP

W większości przypadków algorytm powinien działać bez względu na to, jak zaimplementowano szczegóły poszczególnych funkcji. Możemy zatem w miejsce obiektu wstawić obiekt pochodny i algorytm powinien działać. Oczywiście zachodzi to także w drugą stronę: program który działa dla określonych obiektów, zadziała również dla abstrakcji – czyli obiektów bez zaimplementowanych konkretów. To jest właśnie zasada podstawienia (LSP). W powyższym przykładzie z pionkami możemy zaimplementować losowe ruchy warcabów:

Przykład 12.17: **warcaby5**

```
import random
PRAWO=1
LEWO=-1

class PionekWarcabow(AbsPionek):

  def losowe_ruchy(self):
    while self.y<8:
      self.y +=1
      dx=random.choice([PRAWO, LEWO])
      if (self.x+dx==0) or (self.x+dx>8):
        self.x -= dx
      else:
        self.x += dx
      yield (self.x,self.y)

# dla obiektu klasy AbsPionek program zadziała,
# ale oczywiście nie będzie wykonywał ruchów
pionek = AbsPionek(1,1)
wspolrzedne = [ p for p in pionek.losowe_ruchy()]
print(wspolrzedne)
# ruchy warcabow
pionek = PionekWarcabow(1,1)
wspolrzedne = [ p for p in pionek.losowe_ruchy ()]
print(wspolrzedne)
```

Przykładowe wyjście

```
[]
[(2, 2), (3, 3), (2, 4), (1, 5), (2, 6), (3, 7), (4, 8)]
```

Objaśnienie

Generowanie współrzędnych zadziała poprawnie niezależnie od tego, czy użyjemy obiektu klasy AbsPionek czy PionekWarabow. Dla AbsPionek wygeneruje pustą listę. Dla klasy PionekWarcabow wygeneruje ruchy pionka warcabów (drugi wiersz wyniku).

Warto zauważyć, że metoda **ruch** generuje w klasie abstrakcyjnej wyjątek. Nie da się jej więc użyć tak jak losowe_ruchy. Taka implementacja bez użycia (tylko generująca wyjątek) nie jest też zgodna z zasadami zen Pythona. Uzasadnieniem dla niej byłoby tylko użycie w innej metodzie (jak losowe_ruchy).

W tym miejscu dotykamy bardzo istotnej kwestii związanej z przemysłową produkcją oprogramowania. Załóżmy, że ktoś pisze funkcję drukowania faktury a w zaimportowanym obiekcie nie ma danych dla jednego z obowiązkowych pól. Co powinien zrobić? To zależy od organizacji pracy, a nie tylko ogólnych reguł takich jak SOLID.

ISP, DIP

Te zasady mają szersze zastosowanie w językach w których istnieje silniejsze rozróżnienie interfejsu i klasy. Najprościej rozumieć interfejs jako klasę bez implementacji (z samymi nagłówkami metod). Programista Pythona nie musi sobie tym zbytnio głowy zaprzątać. Warto jednak wiedzieć skąd się to wzięło.

ISP to zasada jest podobna do zasady jednej odpowiedzialności. Zaleca ona rozdrobnienie funkcjonalności tak, by użycie potrzebnego zbioru metod nie wiązało się z dostępem do reszty.

DIP z kolei zaleca korzystanie raczej z interfejsów niż konkretnych klas. W Pythonie można to zredukować do zalecenia,aby wspólne elementy dla klasy bazowej i pochodnej wyodrębnić w trzeciej klasie (interfejsu). Załóżmy, że chcemy mieć metodę losowego przesunięcia pionka. Poniżej przykladowa implementacja:

Przykład 12.18: **warcaby6**

```
1  class IPionek:
2
3    def losuj(self,x,y):
4      raise NotImplementedError()
5
6
7  class AbsPionek:
8    x=y=1
9
10   def __init__(self,x,y):
11      self.x=x
12      self.y=y
13
14   def przesun(self,(x,y)):
15     self.x,self.y = x,y
16
17   def losowe_ruchy(self):
18     return []
19
20
21 import random
22 PRAWO=1
23 LEWO=-1
24
```

```
25 class PionekWarcabow(IPionek, AbsPionek):
26
27   def losuj(self,x,y):
28     if y<8:
29       dx=random.choice([PRAWO, LEWO])
30       if (x+dx==0) or (x+dx>8):
31         dx = -dx
32       return (x+dx,y+1)
33
34   def losowe_ruchy(self):
35     while self.y<8:
36       self.przesun(self.losuj(self.x,self.y))
37       yield (self.x,self.y)
38
39 # ruchy warcabow
40 pionek = PionekWarcabow(1,1)
41 wspolrzedne = [ p for p in pionek.losowe_ruchy ()]
42 print(wspolrzedne)
```

Przykładowy wynik

```
1 [(2, 2), (3, 3), (4, 4), (3, 5), (4, 6), (5, 7), (4, 8)]
```

Objaśnienie

Wykorzystując możliwość dziedziczenia z kilku klas jednocześnie, stworzony została klasa (PionekWarcabow) dziedziczący z klas IPionek i AbsPionek. Pierwsza z nich nie ma żadnej implementacji, a jedynie zdefiniowaną metodę **losuj** (interfejs). Zastosowanie (zgodnie z DIP) interfejsu rzeczywiście upraszcza metodę losowe_ruchy, ale musimy dodatkowo zaimplementować funkcję losuj.

Jak widać daje nam to niewiele i dyskusyjną kwestią jest to, czy nasz kod jest prostszy. Inaczej wyglądałoby to, gdyby metody w interfejsach nie wymagały jakiejkolwiek implementacji (choćby wygenerowania wyjątku).

Kontekst

W obiektach często przechowuje się kontekst przetwarzania danych. Kontekstem może być otwarte połączenie do bazy danych, otwarty plik lub połączenie internetowe. W Pythonie istnieje standardowy sposób obsługi kontekstu. Jest to instrukcja **with**, która była już opisywana w tym podręczniku. Korzystanie z tej instrukcji czyni kod bardziej czytelnym, dlatego jest to zalcane. Na stronie http://book.pythontips.com/en/latest/context_managers.html znajdują się informacje pomocne w tworzeniu menadżerów kontekstu innych niż standardowe.

Pamiętaj, że wszystko jest obiektem

Powyższe zalecenia dotyczyły głównie obiektów jako składowych przy pomocy których budujemy funkcjonalność programu. Czyli obiekt jest traktowany podmiotowo, a dane jakie przetwarza przedmiotowo. W języku Python wszystko jest obiektem – także dane. To właśnie bogactwo środków manipulowania obiektami jest jedną z cech wyróżniających ten język.

Kiedy definiować własne obiekty i metody?

Słowniki, krotki i listy to kolekcje obiektów, które same są obiektami. Obiekty w programach tworzą często bardziej złożone hierarchie (nie tylko na zasadzie obiekt – kolekcja obiektów). Rodzą się więc pytania w rodzaju: kiedy tworzyć własne obiekty, a kiedy korzystać z wbudowanych? Można sformułować kilka zaleceń, które ilustruje poniższy przykład:

Przykład 12.19: **attribs**

```
class User1:
  def __init__(self,ident):
    self.ident = ident
    self.attr = {}

class User2:
  def __init__(self,ident):
    self.ident = ident
    self.name = ''

class User3:
# getter/setter convention
  def __init__(self,ident):
    self.ident = ident

  def setName(self,name):
    self._name = name

  def getName(self):
    return self._name

class User4(object):
# z uzyciem dekoratorow
# https://docs.python.org/2/library/functions.html#property
  def __init__(self,ident):
    self.ident = ident
```

```
27
28    @property
29    def name(self):
30          return self._name
31
32    @name.setter
33    def name(self, value):
34        self._name = value
35
36
37  if __name__ == "__main__":
38    u1=User1('pierwszy')
39    u2=User2('drugi')
40    u3=User3('trzeci')
41    u4=User4('czwarty')
42
43    u1.attr['name']='Nazwa1'
44    u2.name='Nazwa2'
45    u3.setName('Nazwa3')
46    u4.name='Nazwa4'
47    print(u1.attr['name'])
48    print(u2.name)
49    print(u3.getName())
50    print(u4.name)
```

Jeśli to możliwe – użyj właściwości, a nie metod odczytywania zmiany właściwości (getter/setter).

Wbudowane kolekcje danych (słowniki, krotki) mogą zastępować zbiór własności, ale nie warto tego wykorzystywać gdy te własności są złożone (na przykład wartością elementu słownika musiałby być słownik).

Sprawdzanie obecności klucza w słowniku

Obiekt słownika zawiera metodę sprawdzenia, czy zawiera wartość dla danego klucza (`has_key()`). Jednak jej używanie nie jest zalecane. Lepiej użyć metody get z wartością domyślną (drugi parametr):

Przykład 12.20: **dict_keys**

```
d = {'hello': 'world'}
# zamiast:
if d.has_key('hello'):
    print d['hello']    # prints 'world'
else:
    print 'default_value'
# użyj
print d.get('hello', 'default_value') # prints 'world'
print d.get('thingy', 'default_value') # prints 'default_value'
# albo - jeszcze lepiej:
if 'hello' in d:
    print d['hello']
```

W tym ostatnim przypadku wykorzystujemy fakt, że in w przypadku słowników odnosi się do kluczy.

Używaj in gdy to tylko możliwe

Operator in pozwala na sprawdzenie, czy obiekt znajduje się w kolekcji (liście, zbiorze, słowniku), z tym że w przypadku słowników sprawdza się klucze (zob. wyżej). Ale to nie jest jedyne zastosowanie operatora in. Najczęściej używa się go w pętli for. Takie rozwiązanie jest preferowane nawet wówczas, gdy na przykład potrzebujemy indeksu (numeru w liście).

```
colors = ['red', 'blue', 'green']
# Kod nie zalecany:
for i in range(len(colors)):
  print(colors[i])
# Kod zalecany:
for color in colors:
  print(color)
```

Czasem potrzebujemy równocześnie dostępu do elementu listy i jego kolejności (indeksu).

Na przykład mamy nazwy pierwszych czterech cyfr i chcemy je drukować.

Przykład 12.21: **use_in**

```
cyfry = ['zero', 'jden', 'dwa', 'trzy']

# Dla wielu programistów naturalnym będzie rozwiązanie:
n = 0
for cyfra in cyfry:
    print('%s: %s' % (n, cyfra))
    n += 1

# Bardziej „pythonic” jest eliminacja możliwej pomyłki przy zwiększaniu indeksu:

for n in range(len(cyfry)):
    print('%s: %s' % (n, cyfry[n]))

# Jeszcze lepszym rozwiązaniem jest użycie funkcji enumerate:
for (n, cyfra) in enumerate(cyfry):
    print('%s: %s' % (n, cyfra))

# A najprościej zupełnie bez jawnego użycia indeksu:
print(list(enumerate(cyfry)))
```

Wyjście (ostatni wiersz):

```
[(0, 'zero'), (1, 'jden'), (2, 'dwa'), (3, 'trzy')]
```

We wcześniejszym przykładzie z kolorami:

```
#zamiast:
for index in range(len(colors)):
    print('%s: %s' % index,colors[index])
# uzyj funkcji enumerate:
for (index,color) in enumerate(colors):
    print('%s: %s' % index,color)
```

Operator **in** zastępuje także wyszukiwanie (find) w łańcuchach znaków:

```
name = 'Jan Kowalski'
if 'Kowalski' in name:
  print('Tak - to Kowalski!')
# odpowiada
if name.find('Kowalski') != -1:
  print('Tak - to Kowalski!')
```

Łączenie list w łańcuch

Teoretycznie napis (łańcuch) to lista znaków. Jednak to nie są tożsame typy danych. Chcąc zamienić listę na łańcuch znaków posługujemy się metodą join (połącz) obiektu string:

```
letters = ['s', 'p', 'a', 'm']
word = ''.join(letters)
print(letters)
print(word)
```

Teoretycznie można używać do łączenia łańcuchów operatora dodawania (+). Jednak nie jest to efektywne rozwiązanie, więc nie jest zalecane.

```
word = ''
for char in letters:
    word += char
print(word)
```

Stosuj **join** do łączenia znaków i łańcuchów. Jest bardziej efektywna niż dodawanie (+).

Funkcja (metoda) join ma dodatkowy parametr określający separator. Oto przykład jego zastosowania:

```
colors = ['red', 'blue', 'green', 'yellow']
print('Select: %s or %s' %
   (', '.join(colors[:-1]),
   colors[-1]))
```

Oczywiście można też jako parametr funkcji jon użyć wyrażenia for:

```
result = ''.join(fn(i) for i in items)
```

Listy na słownik i z powrotem

Do łączenia dwóch list w słownik służy funkcja zip. Pierwszym parametrem jest lista kluczy, a drugim lista wartości:

Przykład 12.22: **zip_unzip**

```
1 cyfry = [1,2,3,4]
2 nazwy = ['jeden', 'dwa', 'trzy', 'cztery']
3 nazwy_cyfr = dict(zip(cyfry, nazwy))
4 print(nazwy_cyfr)
5 # Powrót jest trywialny:
6 print(nazwy_cyfr.keys())
7 print(nazwy_cyfr.values())
```

Wynik:

```
{1: 'jeden', 2: 'dwa', 3: 'trzy', 4: 'cztery'}
[1, 2, 3, 4]
['jeden', 'dwa', 'trzy', 'cztery']
```

Nie jest ani zalecane ani potrzebne używanie pętli:

```
nazwy_cyfr={}
for i, cyfra in enumerate(cyfry):
   nazwy_cyfr[cyfra] = nazwy[i]
```

Unikaj funkcji map i filter

Mamy dwie funkcje pozwalające tworzyć listy na podstawie innych list

- map()
- filter()

Ich użycie polega na tym, że dla każdego elementu wykonuje się funkcję wyliczającą sprawdzającą, czy element ma się znaleźć w nowej liście (filter) lub zwraca element nowej listy odpowiadający przetwarzanemu elementowi listy starej (map). Trochę to zawiłe, ale przykład wyjaśni wszystko:

```
lista_kwadratow = map(lambda n: n*n, lista_liczba)
```

Zarówno filter jak i map korzysta z funkcji lambda, które jak już zaznaczono nie są zalecanym elementem języka.

Można bez nich obejść się i tym razem. Zamiast funkcji map i filter można użyć wyrażenia for:

Przykład 12.23: **map_filter**

```
a = [3, 4, 5]
# nie zalecane
aa = map(lambda i: i + 3, a)
print(aa)
# zalecane
aa = [i + 3 for i in a]
print(aa)

b = [3, 4, 5]
# nie zalecane
bb = filter(lambda x: x > 4, b)
print(bb)
# zalecane
bb = [i for i in b if i > 4]
print(bb)
```

Unikaj magii

Programiści mówią o „magii”, gdy wykorzystane są mechanizmy inne niż zawarte w standardzie języka. W Pythonie na przykład mamy dostęp do różnych parametrów interpretera. Najczęściej korzystamy ze zmiennej __name__\, która zawiera nazwę modułu. Jest to wykorzystywane do tego aby uwzględnić uruchamianie skryptu zarówno jako odrębnego programu jak i modułu w innym programie. To akurat jest rodzaj „magii” powszechnie akceptowany (a nawet zalecany).

Uzględnij możliwość wykonania i importu skryptu

Uruchomienie programu (głównego skryptu) powoduje ustawienie w zmiennej __name_\ *wartości* ‘_\main__’.

Przykład:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""skic.py

Szkielet modułu do wykonanie z linii komend i po zaimportowaniu"""

def main():
    print('Wykonanie w module:')
    print(__name__)

if __name__ == '__main__':
    print('Uruchomienie jako program (command line)')
    main()
```

Testujemy:

```
1  python szkic.py
2  Uruchomienie jako program (command line)
3  Wykonanie w module:
4  __main__
5
6  python
7  >>> import szkic
8  >>> szkic.main()
9  Wykonanie w module:
10 szkic
```

[v4]

Kilka ważnych kwestii na zakończenie...

Python 2.x czy Python 3.x ?

Którą wersję Pythona wybrać? Najczęściej jesteśmy uwarunkowani dostępnymi bibliotekami. Najpopularniejsze wersja to 2.7 oraz 3.x. Między nimi zachodzi kilka różnic1. Najważniejsze spośród nich2:

1. Instrukcja print staje się funkcją. Nie można więc już w Pythonie 3 napisać: **print 12** można natomiast – tak jak dotąd: **print(12)**
2. Iterator **range** wykorzystywany w pętli for zwracał w Pythonie 2 listę. W Pythonie 3 jest to obiekt zwracający kolejne elementy (taką funkcję pełni w Pythonie 2 xrange). Podobnie jest z funkcją globalną zip, która w Pythonie 2 zwracała listę, a w Pythonie 3 zwraca iterator (generator).
3. W Pythonie 2 do wprowadzania danych służyło input (gdy zachęta była w podwójnych cudzysłowach zwracał napis; inaczej – zależnie od tego wprowadzono). Natomiast raw_input zawsze zwracał łańcuch (string). W Pythonie 3, funkcja raw_input () jest przestarzała, a input zawsze zwraca ciąg znaków (string).
4. Dzielenie liczb w Pythonie 2 zależy od tego, czy liczba jest całkowita. Całkowite (integer) zaokrągla w dół. 5.0/2 == 2,5

   5/2 == 2

   W Pythonie 3 dzielenie bez reszty jest zmiennoprzecinkowe.
5. Inaczej rozwiązano znaki narodowe (w Pythonie 3 napisy są w Unicode – tej kwestii poświęcimy odrębny podrozdział).
6. W Pythonie 3 zyskujemy większe możliwości formatowania wyjścia na konsolę.
7. Kilka konstrukcji syntaktycznych dopuszczalnych w Pythonie 2 stało się w Pythonie 3 obowiązkowych (zob. dalej).

Środowisko wirtualne

Możesz równocześnie mieć dwa środowiska pracy z dwoma różnymi wersjami języka Python. Środowisko definiuje się programem virtualenv[63], który musimy mieć zainstalowany.

Do instalowania może nam posłużyć program pip. Czyli z konsoli Linux wykonujemy (w Windows pip standardowo jest dostępny):

[63] http://programowo.net/2012/11/python-virtualenv-i-pip,27.html

```
sudo pip install virtualenv
```

Sama instalacja wirtualnego środowiska przebiega następująco.

```
virtualenv  -p /usr/bin/python ~/v27py
cd ~/v27py
source bin/activate
```

Gdzie ~/**v27py** podkatalog **v27py** katalogu domowego użytkownika.

Ostatni wiersz to aktywacja środowiska wirtualnego. Od tej pory konsola zgłasza się nazwą środowiska w nawiasach okrągłych. Na przykład:

```
(v27py) janek@localhost:~/v27py$
```

Polecenie **source** jest wewnętrznym poleceniem powłoki Linux[64] (czyli jej wykonanie nie wymaga uruchamiania dodatkowego programu poza interpreterem).

W systemie Windows wykonamy to samo nieco innymi komendami:

```
C:\Python27\Scripts\pip install virtualenv
C:\Python27\Scripts\virtualenv  -p C:\Python27\Scripts\python.exe v27py
v27py\

Scripts\activate
```

Mamy kompletne lokalne środowisko, z ścieżkami dostępu. Możemy doinstalować brakujące pakiety poleceniem pip.

Jeśli nazwy tych pakietów mamy przygotowane w pliku requirements.txt, to instrukcja instalacji ich wszystkich wygląda następująco:

```
pip install -r requirements.txt
```

Kompatybilność

Aby zachować kompatybilność (zgodność, program będzie działał zarówno w wersji 2.x jak i 3.x), używaj zawsze konstrukcji, które są obowiązkowe w Pythonie 3:

[64] http://www.linuxcertif.com/man/1/bash/pl/#WBUDOWANE_POLECENIA_POW%C5%81OKI_86h

1. Nie jest dłużej dopuszczalny zapis obsługi wyjątku bez słowa kluczowego **as**. Piszemy więc try: except Exception as e:
2. Nie dopuszcza się stosowania porównania (relacji „różny”) <> jako synonim do !=. Stosuj więc zawsze zapis !=.

Ponadto używaj:

- print() jako funkcji do drukowania danych
- input() - z zachętą w podwójnych cudzysłowach do wprowadzania danych
- range a nie xrange

Możesz też użyć modułu __future__ lub six do zapewnienia kompatybilności do przodu (czyli dodać do Pythona 2 funkcjonalność funkcjonalność Pythona 3). Wówczas uzyskasz możliwość dzielenia i formatowania takiego jak w Python 3. Na przykład:

```
from __future__ import(
  absolute_import, division, print_function, unicode_literals)

import six
from six.moves import
  (zip, filter, map, range, reduce, input)
```

Poniżej przykład który uruchomiony w Pytonie 2 pokazuje różnice między wersjami:

Przykład:

```
mynumber = 5

print("Python 2:")
print "The number is %d" % (mynumber)
print mynumber / 2,
print mynumber // 2

from __future__ import print_function
from __future__ import division

print('\nPython 3:')
print("The number is {}".format(mynumber))
print(mynumber / 2, end=' ')
print(mynumber // 2)
```

Wyjście:

```
(Python 2:)
The number is 52 2

Python 3:
The number is 52.5 2
```

Objaśnienie:

Początek przykładu pokazuje wyjście instrukcji print oraz dzielenia w Pythonie 2. Dalszy fragment – po zaimportowaniu __future__ daje wynik taki, jakbyśmy używali Pythona 3.

Polskie znaki

Stosowanie tekstów zawierających inne litery niż łacińskie (w polskim języku ą, ę , ł itd....) bywa przyczyną wielu problemów. Dla programistów zaczynających przygodę z Pythonem może to być zaskoczenie: program przerywa się napotykając „polską" literę. Jednak bardzo restrykcyjne podejście do zasad kodowania takich znaków wymusza poprawne używanie napisów.

Przede wszystkim należy konsekwentnie stosować kodowanie UTF8 (Unicode). Ono nie jest domyślne i dlatego rozpoczynamy moduł od zapisu:

```
# -*- coding: utf-8 -*-
```

Dzięki temu możemy w stałych i komentarzach stosować literki z ogonkami...

W tym miejscu trzeba zwrócić uwagę na to, że w Pythonie3 łańcuchy znaków są domyślnie kodowane w Unicode, a w Pythonie 2 tylko wtedy, gdy poprzedzimy je literką u:

```
>>> napis='łąka'
>>> unapis=u'łąka'
>>> bnapis=b'łąka'
>>> bnapis==napis
True
>>> napis==unapis
__main__:1: UnicodeWarning: Unicode equal comparison failed to convert both argu\
ments to Unicode - interpreting them as being unequal
False
>>> napis.decode('utf8')==unapis
True
>>> bnapis==unapis.encode('utf8')
True
```

Poza użyciem konstrukcji u"" (unicode) widzimy w tym przykładzie użycie b"" - co oznacza zapis „binarny" łańcucha (taki jakim jest – bez konwertowania na Unicode). Widać też, że domyślnie napis jest binarny i użycie go jako unicode powoduje błąd. Aby uniknąć błędu – musimy zdekodować napis na napisu z polskimi znakami przy pomocy metody decode (parametrem jest sposób kodowania – gdyż nie jest zdefiniowane domyślne kodowanie). Dostępna jest także odwrotna metoda (encode).

W Pythonie 3 mamy odwrotną sytuację – domyślnie napisy nie są binarne:

```
>>> napis='łąka'
>>> unapis=u'łąka'
>>> bnapis=b'łąka'

File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.

>>> bnapis=unapis.encode('utf8')
>>> napis==unapis
True
>>> napis==bnapis
False
>>>
```

Rozwiązanie z Pythona 3 jest dużo wygodniejsze – bo konsekwentne stosowanie Unicode pozwala zapomnieć o problemach z polskimi znakami. Można to rozwiązanie stosować w Pythonie 2 przy __future poznanego wcześniej:

```
from __future__ import unicode_literals
```

Nawet po przejściu na Python3 trzeba jednak pamiętać o kodowaniu binarnym – na przykład przy tworzeniu szyfrów (identyczny napis da różny kod zależnie od szyfrowania) lub gdy napis w bazie danych jest z innym kodowaniem (wówczas trzeba odpowiednio skonfigurować połączenie do bazy i ewentualnie umiejętnie stosować przy odczycie / zapisie code i decode).

Uwagi dla osób programujących w innych językach

Łatwość programowania w języku Python jest związana między innymi z tym, że do rozwiązywania każdego problemu używa się możliwie najprostszych konstrukcji. Celem programisty jest precyzyjnie opisać co ma być zrobione w programie. Nic więcej. Stąd pomysł na odmienną niż w innych językach konstrukcję podstawowej pętli (**for**).

Chcąc wykonać jakąś operację na trzech elementach w większości języków napiszemy cos w rodzaju:

```
for (i=0; i<3; i++) {
```

Czyli musimy użyć indeks i – nawet gdy nie jest nam potrzebny. W Pythonie piszemy:

```
for element in lista:
```

Indeks i wprowadzamy dopiero, gdy jest niezbędny (przy pomocy generatora range).

Bardzo rzadko używa się pętli innych niż for. Ma to dodatkową zaletę: program możemy czytać sekwencyjnie. Bo przecież taka pętla to tylko zapis równoważny z „zrób coś dla wszystkich elementów zbioru". Nie ma problemu „zapętlenia" (o ile zbiory nie są nieskończone).

Pozostałe różnice nie są tak znaczące, ale przechodząc do programowania w Pythonie, warto zdawać sobie z nich sprawę.

Typy danych

Jeśli chcemy dodać dwie zmienne, to muszą one być zgodnych typów. Błędne wyrażenie:

```
>>> a='2'
>>> b=2
>>> a+b
```

Taki sposób typowania (wiązania zmiennej z typem danych nazywa się silnym.

Poza wiązaniem silny/słaby rozróżniamy wiązanie statyczne (typ zmiennej się nie zmienia) i dynamiczne (zmienna przyjmuje typ wartości która do niej wstawiamy). W Pythonie mamy wiązanie dynamiczne. To się wykona poprawnie (ciąg dalszy poprzedniego przykładu):

```
>>> b=int(b)
>>> a+b
```

Typowanie	Słabe	Silne
Statyczne	C, C++	Java, Pascal
Dynamiczne	Perl, VB	Python

Typowanie statyczne odbywa się poprzez deklarację (jakiego typu zmienna ma być). Dynamiczne typowanie odbywa się poprzez użycie (zależy jaką daną wstawimy inicjując zmienną).

Wśród prostych typów trzeba zwrócić uwagę na:

- logiczne **True** i **False** - równoważne wartości **1** i **0** odpowiednio;

- napisy w wielu wierszach (różnią się trzema apostrofami zamiast jednym);

Typy złożone (listy, krotki) nie są płaskie ani jdnordne! Mogą w liście znajdować się elementy różnych typów – w tym inne listy!

Przekazywanie zmiennych do funkcji

Rozwiązanie Pythona jest tyleż intuicyjnie proste, co nietypowe. Jeśli typ zmiennej jest prosty – to jeśli użyjemy jej jako argumentu funkcji - przekazywana jest wartość (tak samo nazywająca się zmienna wewnątrz funkcji jest inną zmienną niż ta zewnętrzna). W przypadku danych złożonych przekazywany jest wskaźnik.

Pewnym „dziwactwem" języka jest inicjowanie parametrów domyślnych funkcji chwili definiowania funkcji, a nie jej wywołania Załóżmy, że potrzebujemy funkcji, która albo dodaje przekazany element do listy przekazanej w drugim parametrze, albo zwraca listę złożoną z jednego elementu:

```
1  def bad_append(new_item, a_list=[]):
2      a_list.append(new_item)
3      return a_list
```

Ten przykład jest nieco trudniejszy od innych w tym podręczniku – ale warto go przeanalizować, by uniknąć błędów i lepiej zrozumieć specyfikę języka. Przy pierwszym wywołaniu funkcji z jednym tylko parametrem użyta zostanie wartość domyślna (pusta lista). Tu nie ma zaskoczenia:

```
1  >>> print bad_append('one')
2  ['one']
```

Jednak przy drugim wywołaniu domyślną wartość już mamy nadaną, a ponieważ jest to zmienna złożona mamy niespodziankę:

```
1  >>> print bad_append('two')
2  ['one', 'two']
```

Właściwy sposób, aby otrzymać listę domyślną (tu: pustą) przy braku parametru jest stworzenie jej w czasie wykonywania - wewnątrz funkcji:

```
def good_append(new_item, a_list=None):
    if not a_list:
        a_list = []
    a_list.append(new_item)
    return a_list
```

Przy typach prostych nie ma takiego problemu:

```
def sumator(item, sum=0):
    return sum+item
```

Sprawdzamy:

```
>>> sumator(1)
1
>>> sumator(9)
9
>>> suma=11
11
>>> print sumator(6,suma)
17
```

Instrukcje warunkowe

W Pythonie nie ma instrukcji przełączania **switch** (case). Można użyć w to miejsce

```
if..elif..else ...
```

Klauzula **else** jest także dostępna w pętlach **while**, ale nie zaleca się używania takiej konstrukcji (innego użycia **else** niż jako części **if**).

Polecane linki

1 http://www.w3ii.com/pl/python3/python3_whatisnew.html[65] http://www.diveintopython3.net/porting-code-to-python-3-with-2to3.html[66]

2W zob. też http://python-future.org/compatible_idioms.html[67]).

[v2]

[65] http://www.w3ii.com/pl/python3/python3_whatisnew.html
[66] http://www.diveintopython3.net/porting-code-to-python-3-with-2to3.html
[67] http://python-future.org/compatible_idioms.html