Python w Elektronicznej Sieci #4: Wprowadzenie do programowania w Pythonie (3/3)

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2020-03-17

1 Zmienne i ich typy

1.1 Funkcje jako argumenty funkcji

W powyższym przykładzie jednym za argumentów funkcji `sorted()` jest inna funkcja. Zauważ, że funkcja może być takim samym argumentem innej funkcji jak dowolna inna zmienna, może być też wynikiem zwracanym przez funkcję oraz może być przechowywana w zmiennej.

```
def dzialanie(operacja):
    if operacja == "dodaj":
        def f(a, b):
            return a+b
        return f
    elif operacja == "mnóż":
        def f(a, b):
            return a*b
        return f
def dwa(funkcja, argument):
    return funkcja(2, argument)

d = dzialanie("dodaj")
a = dwa(d, 11)
b = dzialanie("mnóż")(3,4)
print(a, b, d(3,4))
```

```
13 12 7
```

Zauważ że:

- wynikiem funkcji `dzialanie()` jest funkcja wykonująca wskazane działanie,
- funkcja `dwa()` jako argumenty przyjmuje funkcję realizującą działanie dwuargumentowe i jeden argument przekazywany do niej,
- zmienna d wskazuje na funkcję zwróconą przez funkcję `dzialanie()` i może być używana jako funkcja.

1.2 Klasy i struktury

Inną metodą grupowania zmiennych i funkcji jest definiowanie własnych klas:

```
class NazwaKlasy:
  # pola składowe
  a, d = 0, "ala ma kota"
  # metody składowe
  def wypisz(self):
    print(self.a + self.b)
  # metody statyczna
  def info():
    print("INFO")
  # konstruktor (z jednym argumentem)
  def __init__(self, x = 1):
    # i kolejny sposób na utworzenie pola składowego klasy
    self.b = 13 * x
```

Warto zauważyć jawny argument metod składowych klasy w postaci obiektu tej klasy. Możliwe jest także dziedziczenie po jednej lub kilku klasach bazowych, w tym celu definicje klasy rozpoczynamy:

```
class NazwaKlasy(Bazowa1, Bazowa2):
```

Tworzenie obiektu klasy i używanie go:

```
k = NazwaKlasy()
k.a = 67
k.wypisz()
```

```
80
```

Obiekty można rozszerzać o nowe składowe i funkcje:

```
k.c = k.a + 10
print(k.c)
```

```
77
```

W ten sposób można też tworzyć całe struktury:

```
class Pusta():
  pass
x = Pusta()
x.a = 3
x.b = 4
```

Od strony implementacyjnej są one trzymane w słowniku związanym z danym obiektem o nazwie `__dict__`.
Spróbuj wypisać zawartość `x.__dict__` oraz `k.__dict__`.

Do metod klasy możemy odwoływać się także z podaniem nazwy klasy a nie obiektu, w takim wypadku jeżeli nie są to metody statyczne należy przekazać jako argument obiekt danej klasy lub go udający[1]:

```
NazwaKlasy.info()
NazwaKlasy.wypisz(k)
NazwaKlasy.wypisz(x)
```

```
INFO
80
7
```

1.3 Iteratory i generatory

Iterator jest obiektem pozwalającym na dostęp do kolejnych elementów jakiejś kolekcji (np. listy). Są one przydatne np. gdy chcemy uzyskiwać kolejne elementy kolekcji nie iterując po niej w ramach pętli **for**. Jego użycie wygląda następująco:

```
l = [6, 7, 8, 9]
i = iter(l)  # zmienna i jest tutaj iteratorem
print( next(i) )
print( next(i) )
```

Niekiedy zamiast tworzenia listy lepsze może być uzyskiwanie jej kolejnych elementów "na żywo". Funkcjonalność taką w pythonie zapewniają generatory. Są to funkcje które zwracają kolejne elementy danej kolekcji używając słowa kluczowego **yield**, zamiast **return**. Pamiętają one też swój stan wewnętrzny pomiędzy wywołaniami w ramach poszczególnych iteracji.

Generatory możemy używać np. do iterowania po nich w pętli **for**, możemy tez używać iteratorów do pobierania kolejnych wartości z generatora:

```
def f(l):
    a, b = 0, 1
    for i in range(l):
```

1. Wystarczy żeby taki obiekt miał metody i składowe używane przez dana metodę, nie musi to być obiekt tej klasy.

```
        r, a, b = a, b, a + b
        yield r

ii = iter( f(8) )
for i in f(16):
    print("i =", i)
    if i > 6:
        print("ii =", next(ii))
```

Można także tworzyć generatory nieskończone:

```
def ff():
    a, b = 0, 1
    while True:
        r, a, b = a, b, a + b
        yield r
```

1.4 Pliki

Do tej pory wszystkie dane, z których korzystały nasze programy, wprowadzaliśmy bezpośrednio do kodu programu. W realnych zastosowaniach bardzo często użyteczniejsze jest korzystanie z danych zapisanych w osobnych plikach.

1.4.1 Zapisywanie tekstu do pliku

Zapis do pliku tekstowego możemy zrealizować w sposób następujący:

```
plik = open('dane.txt', 'wt', encoding='utf8')
plik.write("teskt1\n")
plik.write("teskt2\nteskt3")
plik.close()
```

Jak to działa?

- Polecenie z pierwszej linijki otwiera plik `dane.txt` i zapewnia dostęp do niego poprzez zmienną `plik`. Opcja `'w'` oznacza, że plik jest otwarty „do zapisu” (od angielskiego *write*). Opcja `'t'` oznacza, że plik traktowany jako plik tekstowy[2]. Argument `encoding` pozwala na określenie kodowania użytego do zapisu pliku tekstowego, jest on opcjonalny i gdy nie zostanie podany kodowanie pliku zależne jest od ustawień systemowych.
- Druga i trzecia komenda zapisuje podany jako argument tekst do pliku `dane.txt` (zwróć uwagę na wstawianie nowej linii przy pomocy `'\n'`)
- Ostatnie polecenie zamyka dostęp do pliku `dane.txt`.

Po uruchomieniu powyższego kodu powinien zostać utworzony plik „dane.txt”, zawierający 3 linie tekstu. Jeżeli plik taki wcześniej istniał zostanie on nadpisany.

1.4.2 Wczytywanie tekstu z pliku

```
plik = open('dane.txt', 'rt', encoding='utf8')
for linia in plik:
  print(linia, end="")
plik.close()
```

2. Tekst możemy zapisywać także do plików otwieranych jako binarne, w takim wypadku argument funkcji write musi mieć typ `bytes` a nie `str`, czyli być już jawnie zakodowanym w jakimś standardzie.

Zauważ, że została używa opcja `'r'` do otwarcia pliki co oznacza otwarcie do odczytu (od angielskiego *read*). Jeżeli chcemy wczytać cały plik do zmiennej napisowej możemy, zamiast pętli czytającej kolejne linie, użyć metody `read()`:

```
plik = open('dane.txt', 'rt', encoding='utf8')
napis = plik.read()
plik.close()
```

1.4.3 Czekanie na dane

Niekiedy nasz program musi poczekać na jakieś dane (np. wprowadzane z standardowego wejścia przez użytkownika). Typowo funkcje odczytu (takie jak `sys.stdin.read()`, `sys.stdin.readline()`, `input()`) czekają na koniec wczytywanych danych lub na koniec linii. Komplikacja pojawia się kiedy chcielibyśmy aby nasz program miał ograniczenie czasowe takiego oczekiwania lub czekał na pojawienie się danych w jednym z kilku źródeł. W takich przypadkach przydatna jest funkcja systemowa `select()`, którą w Pythonie znajdziemy w module *select*.

```
import sys, os, select

rdfd, _, _ = select.select([sys.stdin], [], [], 3.0)

if not rdfd:
        print("czas minął")

for fd in rdfd:
        print("czytam z:", fd)
        a = os.read(fd.fileno(), 1024)
        print("wczytałem:", a)
```

Funkcja `select()` przyjmuje 3 listy „deskryptorów plików” (czyli tego co zwraca np. funkcja `open()`) oraz ilość sekund, którą ma czekać na początek danych. Pierwsza lista związana jest z plikami z których chcemy czytać, druga pisać, a trzecia z plikami na których czekamy na wyjątkowe warunki. Funkcja ta zwraca również 3 takie listy, ale zawierające jedynie deskryptory plików na których pożądana operacja jest możliwa (np. są dane do wczytania, można zapisać dane).

Funkcja `select()` kończy działanie gdy pojawią się jakiekolwiek dane (nie czeka na koniec danych – EOF). Zauważ, że do odczytu zastosowana została funkcja `os.read()` a nie metoda `fd.read()`, wynika to z faktu, iż `fd.read()` czeka na EOF lub podaną ilość bajtów, a `os.read()` wczytuje to co jest dostępne i ogranicza jedynie maksymalną ilość wczytywanych danych (resztę możemy doczytać kolejnym wywołaniem).

2 Podstawy programowania równoległego

2.1 procesy i fork()

Aby w systemie mógł działać więcej niż 1 proces konieczna jest możliwość utworzenia nowego procesu (potomka) z poziomu procesu aktualnie działającego (rodzica). Możliwe są dwa podejścia:

- utworzenie ”czystego” procesu uruchamiającego podany kod programu z podanymi argumentami (spawn)
- utworzenie kopii aktualnego procesu, która zacznie wykonywać się niezależnie od momentu rozgałęzienia (fork)

W przypadku zastosowania fork proces potomny otrzymuje kopię pamięci rodzica (ma dostęp do wszystkich jego zmiennych oraz zasobów uzyskanych przed fork(); dalsze operacje na zmiennych są niezależne). Po utworzeniu kopi procesu można (ale nie trzeba) zastąpić wykonywany w nim program innym poprzez funkcje z rodziny exec. Cechy te powodują że mechanizm fork jest bardziej elastyczny od spawn.

```
import os

print("nasz pid to:", os.getpid())

pid = os.fork()
if pid == 0:
        print("jesteśmy w procesie potomnym, nasz pid to:", os.getpid())
else:
        print("jesteśmy w rodzicu, pid naszego potomka to:", pid)
```

2.2 wywołanie zewnętrznej komendy

Najprostszym sposobem uruchomienia innej komendy z poziomu Pythona jest użycie funkcji `system()` z modułu os:

```
import os

inStr = "Ala ma kota\nKot ma psa\n..."

os.system('echo -en "' + inStr + '" | grep -v A')
```

Jak widać przekazujemy do niej napis takiej samej postaci jak wyglądałby komenda uruchamiana w terminalu. Mechanizm ten nie daje jednak zbyt dużej kontroli nad uruchamianiem tego polecenia (nie pozwala na proste odebranie jego standardowego wyjścia, przekazanie wejścia również wymaga dodatkowego zabiegu w postaci dodania komendy echo, itd.). Bardziej elastycznym rozwiązaniem jest pythonowy moduł *subprocess*:

```
import subprocess

inStr = "Ala ma kota\nKot ma psa\n..."

# uruchamiamy subprocess z grep'em
res = subprocess.run(["grep", "-v", "A"], input=inStr.encode(),
 ↪  stdout=subprocess.PIPE)
print("Kod powrotu to: " + str(res.returncode))
print("Standardowe wyjście z komendy to: " + res.stdout.decode())
# warto zwrócić uwagę na kodowanie i dekodowanie napisów
# (przekazywanych/odbieranych przez stdin/stdout) do / z utf-8

# jeżeli chcemy korzystać np. z znaków uogólniających powłoki lub podać
# komendę jako pojedynczy napis (a nie listę argumentów) to można użyć
# opcji shell=True:
subprocess.run(["ls -ld /etc/pa*"], shell=True)
# jeżeli potrzebujemy tylko rozbicia napisu na listę argumentów można
# użyć shlex.split()

# run() pozwala także (obok subprocess.PIPE) na przekazywanie
```

```
# istniejących deskryptorów (lub subprocess.DEVNULL, co ignoruje wyjście)
# w ramach stdin, stdout, stderr

# moduł subprocess oferuje także funkcję Popen() dającą większą kontrolę
# nad uruchamianiem komendy
```

2.3 komunikacja międzyprocesowa

W systemie wieloprocesowym konieczne jest zapewnienie mechanizmów komunikacji pomiędzy procesami, zwłaszcza jeżeli grupa procesów ma realizować wspólne zadanie.

Jednym z takich mechanizmów (można powiedzieć że nawet podstawowym) jest poznane już wcześniej łącze nie nazwane (pipe, uzyskiwane np. w bashowej linii poleceń przy pomocy `|`) pozwalające na przekazywanie strumienia danych od jednego do kolejnego procesu. Podobnie działa łącze nazwane z tym że nie jest uzyskiwane w wyniku funkcji `pipe()` a otwarcia specjalnego pliku (utworzonego `mkfifo()`) przez dwa procesy (jeden do czytania drugi do pisania).

Innymi mechanizmami komunikacji międzyprocesowej są m.in:

- sygnały
- kolejki komunikatów
- pamięć współdzielona

Stosowanie pamięci współdzielonej wymaga często też stosowania mechanizmów ochrony dostępu do niej (wejścia do „krytycznych sekcji" kodu). Koncepcja takiej ochrony wygląda następująco:

```
if !blokada:
        blokada = True
        # działania na pamięci wspólnej
        blokada = False
```

Jednak nie może być zrealizowana w tak prosty sposób, gdzyż przełączenie procesów może nastąpić pomiędzy sprawdzeniem warunku na zmiennej `blokada` a zmianą jej wartości (lub mogą one działać idealnie równolegle i w tym samym momencie sprawdzać wartość zmiennej `blokada`). Dlatego do ochrony sekcji krytycznych stosuje się mechanizmy systemowe takie jak semafory i lock'i.

2.4 wątki

Oprócz możliwości pełnego rozgałęzienia procesu (utworzenia potomka), możliwe jest także tworzenie wątków (zwanych też lekkimi procesami) w ramach bieżącego procesu. Wątek (w odróżnieniu od procesu potomnego) korzysta z tej samej pamięci (przestrzeni adresowej) co oryginalny proces i wszystkie inne jego wątki (czyli *out of the box* mają pamięć współdzieloną). Jednak każdy wątek posiada niezależny stos (umieszczany w innym fragmencie współdzielonej pamięci), który jest używany m.in. do przechowywania zmiennych lokalnych (w tym argumentów funkcji), czyli dopóki ograniczamy się do zmiennych lokalnych nie ma potrzeby stosowania ochrony sekcji krytycznych ze względu na dostęp do pamięci.

2.5 „Python-way"

Zaprezentowane powyżej podejście korzysta w dużej mierze z funkcji analogicznych do funkcji systemowych biblioteki standardowej C zgromadzonych w module *os*. Python oferuje obok wspomnianego modułu *subprocess* także inne własne mechanizmy związane z tworzeniem wielu procesów poprzez moduł *multiprocessing* oraz oferuje wsparcie dla wątków w module *threading*[3]. Jednak, jako że w ramach tego kursu nie będziemy zajmować się programowaniem równoległym jako takim, to modułów tych nie omówimy w

3. Należy mieć na uwadze iż pythonowe wątki są niepełnowartościowe - ze względu na konstrukcję interpretera CPython, jedynie jeden wątek w danej chwili może być aktywny - wykorzystywać CPU, pozostałe mogą jedynie czekać.

tym skrypcie ani na zajęciach. Zainteresowanym polecam zapoznanie się z http://vip.opcode.eu.org/#Procesy_i_wątki.

3 Biblioteki

Ideą korzystania z funkcji w trakcie tworzenia programu jest zapewnienie jego większej czytelności oraz unikanie powtarzania kodu robiącego to samo w wielu miejscach programu – kod umieszczamy w funkcji którą tylko wywołujemy z odpowiednimi argumentami i odbieramy wynik działania (np. poprzez zwracaną wartość). Rozwinięciem tej ideii są biblioteki stanowiące zbiory funkcji oraz struktur danych (własnych typów zmiennych) służących do realizacji określonych zadań.

Do tej pory korzystaliśmy z elementów standardowej biblioteki dostarczanej z Pythonem. W rozdziale tym zaprezentujemy kilka różnych przykładowych bibliotek (w tym wchodzących w skład biblioteki standardowej Pythona), jednak żadnej z nich nie będziemy tutaj szczegółowo omawiać, gdyż nie miałoby to większego sensu. Istnieje ogromna liczba bibliotek dedykowanych różnym celom (obsługa formatów plików, standardów komunikacyjnych, tworzenie grafiki, ...) i nie ma sensu uczyć się ich bez realnej potrzeby zastosowania – programowanie w dużej mierze polega na wyszukiwaniu właściwych bibliotek, zapoznawaniu się z ich dokumentacją i wykorzystywaniu ich w własnych programach. W przypadku Pythona biblioteki najczęściej mają postać modułów pythonowych, które włączamy poprzez deklarację `include`.

3.1 XML

Extensible Markup Language (XML) jest tekstowym formatem wymiany danych. W odróżnieniu od formatu klasycznego formatu utożsamiającego linię z rekordem złożonym z pól oddzielanych wskazanym separatorem może on w łatwy sposób opisywać bardziej złożoną (drzewiastą a nie tabelkową) postać danych. Dokument XML składa się z zagnieżdżonych w sobie znaczników, każdy z nich może posiadać atrybuty oraz wartość, którą jest tekst zawierający lub nie kolejne znaczniki. Kolejność występowania elementów w dokumencie jest znacząca. Każdy znacznik otwierający posiada odpowiadający mu znacznik zamykający (np. aa), znaczniki bez wartości mogą być samo-zamykające (np. <g />). Dokumenty HTML mogą być zgodne z wymogami formalnymi XML tym samym stanowiąc dokumenty XML.

Do obsługi XML w Pythonie można skorzystać np. z modułu *ElementTree* (ale nie jest on jedyną biblioteką której możemy użyć):

```
import xml.etree.ElementTree as xml

txt = """<a>
        <b>A<h>qwe ... rty</h></b> ABCD... &amp;&apos; HIJ...
        <c x="q" w="p p">EE FĄ</c> <g y="zz" />
        <c x="pp">123 <d rr="oo">456</d> 78 90.</c>
</a>"""

rootNode = xml.fromstring(txt)

print("nazwa głównego elementu to:", rootNode.tag)
print("jego potomkowie to:")
for subNode in rootNode:
        print(" ", subNode.tag, ":", xml.tostring(subNode, encoding="unicode"))

# możemy pobrać listę potomków o określonej nazwie
# albo od razu po nich iterować pętlą for subNode in rootNode.iter("c"):
cSubNodes = list( rootNode.iter("c") )
if cSubNodes:
        for subNode in cSubNodes:
                print('element "c" ma atrybuty': subNode.attrib
```

```
else
        print('nie ma elementów "c"')

# możemy też używać iteratorów bezpośrednio, np:
print("pierwszy węzeł c ma atrybuty:")
try:
        ci = rootNode.iter("c")
        print(next(ci).attrib)
except StopIteration:
        print(" [brak takiego węzła]")
```

ElementTree pozwala też na modyfikowanie XMLa poprzez zmianę/dodawanie/usuwanie atrybutów, czy też całych tagów.

Innym sposobem zapisu ustrukturyzowanych danych w postaci tekstowej jest JSON. Przypomina on trochę output funkcji print z podanym do niej słownikiem lub listą. Do jego obsługi w Pythonie służy moduł *json*:

```
import json, pprint

a='''{
        "info": "bbb",
        "ver": 31,
        "d": [
                {"a": 21, "b": {"x": 1, "y": 2}, "c": [9, 8, 7]},
                {"a": 17, "b": {"x": 6, "y": 7}, "c": [6, 5, 4]}
        ]
}'''

# interpretacja napisu jako zbioru danych w formacie json
d = json.loads(a)

# wypisanie zbioru danych
pprint.pprint(d) # pprint ładnie formatuje złożone zbiory danych

# jak widać jest to zagnieżdżona struktura list i słowników
# odpowiadająca 1 do 1 temu co było w napisie

# dostęp do poszczególnych elementów: "po pythonowemu"
print(d["d"][1]["b"])
d["d"][1]["b"]["x"] = "XXX"

# wygenerowanie json'a w oparciu o zmienną pythonową
c = json.dumps(d, ensure_ascii=False)
print(c)
```

3.2 SQL

Innym sposobem przechowywania danych niż w postaci plików tekstowych są systemy baz danych. Standardowym językiem używanym do komunikacji z systemami bazodanowymi jest SQL. Pomimo jego standaryzacji istnieją różnice w składni zapytań dla poszczególnych silników bazodanowych (takich jak: MariaDB, PostgreSQL, SQLite, ...).

Typowo komunikacja z bazą danych odbywa się za pośrednictwem biblioteki odpowiedzialnej za nawiązanie połączenia z serwerem i przekazywanie do niego zapytań SQL. Wymaga to działania osobnego procesu (często nawet na innej maszynie) obsługującego silnik bazodanowy, co jest pożądanym rozwiązaniem dla baz danych z których równocześnie może korzystać wielu klientów. Typowym przykładem może być komunikacja skryptów jakiegoś serwisu interetowego z bazą danych.

Jednak takie podejście nie jest wygodne w rozwiązaniach nie wymagających współdzielenia bazy danych. Do zastosowań takich można użyć biblioteki SQLite, która pozwala na łatwe stosowanie bazy SQLowej do wewnętrznych potrzeb aplikacji, bez konieczności uruchamiania osobnego systemu bazodanowego. SQLite można wykorzystywać także bezpośrednio z poziomu Pythona, dzięki modułowi *sqlite3*:

```
import sqlite3
import os.path

if os.path.isfile('example.db'):
        create = False
else:
        create = True

conn = sqlite3.connect('example.db')
c = conn.cursor()

if create:
        print("create new db")
        c.execute("CREATE TABLE users (uid INT PRIMARY KEY, name TEXT);")
        c.execute("CREATE TABLE posts (pid INT PRIMARY KEY, uid INT, text TEXT);")

        c.execute("INSERT INTO users VALUES (21, 'user A');")
        c.execute("INSERT INTO users VALUES (2671, 'user B');")

        c.execute("INSERT INTO posts VALUES (1, 21, 'abc ..');")
        c.execute("INSERT INTO posts VALUES (2, 21, 'qwe xyz');")
        c.execute("INSERT INTO posts VALUES (3, 2671, 'test');")

        conn.commit()

maxUid = 100
for r in c.execute("SELECT * FROM users WHERE uid < ?;", (maxUid,)):
        print(r)

for r in c.execute("SELECT u.name, p.text FROM users AS u JOIN posts AS p ON (u.uid
 ↪  = p.uid);"):
        print(r)
```

3.3 GUI

Przykłady użycia 3 różnych graficznych interfejsów użytkownika z poziomu Pythona można znaleźć na http://vip.opcode.eu.org/#Graficzny_interfejs_użytkownika. W odróżnieniu od poprzednich przykładów, te biblioteki nie wchodzą w skład pythonowskiej biblioteki standardowej i mogą wymagać doinstalowania odpowiednich pakietów oprogramowania.

4 Zadania

Zadanie 4.0.1

Napisz funkcję, który wczytuje dane z standardowego wejścia. Funkcja powinna przyjmować jeden argument określający maksymalny czas oczekiwania na kolejną porcję danych. Każde pojawienie się danych wejściowych powinno resetować odliczanie timeoutu podanego w argumencie. Po skutecznym upływie tego timeoutu funkcja powinna zwrócić wszystkie wczytane dane.
Wskazówka: zmodyfikuj przykład użycia funkcji select() podany w skrypcie.

Zadanie 4.0.2

Napisz program który utworzy 1 potomka, rodzić powinien wypisać PID potomka i swój. Natomiast potomek powinien utworzyć kolejny proces w którym zostanie uruchomiona komenda `ps -Al` w taki sposób aby potomek odebrał do zmiennej jej standardowe wyjście i wypisał je na ekran.

Zadanie 4.0.3

Napisz funkcję która przyjmuje dwa argumenty: listę oraz funkcję. Funkcja ma za zadanie wykonać przekazaną do niej funkcję na każdym elemencie listy. Przykład użycia:

```
>>> wykonaj([1,2,3], print)
1
2
3
```

Zadanie 4.0.4

Napisz funkcję która przyjmuje dwa argumenty: słownik oraz nazwę pliku. Funkcja ma utworzyć plik o podanej nazwie i zapisać do niego otrzymany słownik, w taki sposób że każda linii odpowiada jednej parze klucz wartość, a separatorem pomiędzy kluczem a wartością jest znak tabulacji. Dla uproszczenia zakładamy że elementy słownika są napisami (zarówno klucze jak i wartości) i nie zawierają znaków nowej linii ani tabulacji.

5 Literatura dodatkowa

- *The Python Tutorial* (https://docs.python.org/3/tutorial/) - oficjalny Tutorial Pythona.
- *Vademecum informatyki praktycznej* (http://vip.opcode.eu.org/) - zbiór materiałów na temat elektroniki i programowania m.in. w Pythonie, wykorzystywany m.in. w edycji VIIIbis MdCŚ.
- *Biblioteka Riklaunima: Podstawy Pythona* (http://www.python.rk.edu.pl/w/p/podstawy/).
- *A Byte of Python* (https://python.swaroopch.com/).
- *How to Think Like a Computer Scientist: Learning with Python 3* (http://openbookproject.net/thinkcs/python/english3e/).
- *Zanurkuj w Pythonie* (https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie).

6 Praca domowa

6.1 Instrukcja wysyłania rozwiązań

Rozwiązania zadań domowych należy przesłać na adres licealisci.pracownia@icm.edu.pl wpisując jako temat wiadomości `g2.x PD2`, gdzie `x` to numer grupy, np. `g2.1 PD2` dla grupy nr. 1, itd. Zadania domowe są nie obowiązkowe, jednak zachęcamy do ich robienia i wysyłania rozwiązań (nawet niekompletnych). Termin nadsyłania zdań domowych to 2020-03-03 godz. 16^{00}.

Na ten adres można także nadsyłać ewentualne pytania do zadań (zarówno domowych jak i innych zamieszczonych w skrypcie), w tym wypadku także prosimy o umieszczenie w temacie wiadomości `g2.x`, gdzie `x` to numer grupy.

6.2 Zadania domowe

Zadanie 6.2.1 — 1 pkt

Napisz program który utworzy 1 potomka, potomek powinien wypisać PID rodzica i swój.

Zadanie 6.2.2 — 2 pkt

Korzystając z metod klasy `list` i/lub funkcji `sorted()` napisz funkcję która sortuje podaną listę w kolejności malejącej.

Zadanie 6.2.3 — 3 pkt

Napisz funkcję która konwertuje listę napisów postaci `klucz=wartosc` na słownik. Funkcja musi dokonywać podziału napisów z listy w oparciu o pierwsze wystąpienie znaku równości przy pomocy metody `find()` typu przechowującego napisy (`str`). Funkcja musi dodawać kolejne napisy do słownika w taki sposób że część przed znakiem równości stanowi klucz, a część po znaku równości stanowi wartość.
Np. dla listy postaci: `["aa=13", "b=Ala=kot", "f=xyz"]` funkcja powinna zwrócić słownik:

```
{'b': 'Ala=kot', 'aa': '13', 'f': 'xyz'}
```