Linux i Python w Elektronicznej Sieci #03: Python bardziej zaawansowany i biblioteki

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

2021-03-18

1 Zmienne i ich typy

1.1 Określanie typu zmiennej

Do tej pory poznaliśmy kilka typów zmiennych w Pythonie: liczby, napisy oraz listy. Poznaliśmy także metody konwersji pomiędzy niektórymi z typów (np. instrukcje `str()`, `int()`). Jeżeli chcemy dowiedzieć się jakiego typu jest dana zmienna możemy skorzystać z funkcji `type()`:

```
a, b, c = 1, 3.14, "Python"
print(a, type(a))
print(b, type(b))
print(c, type(c))
c = (a == 1)
print(c, type(c))
```

```
1 <class 'int'>
3.14 <class 'float'>
Python <class 'str'>
True <class 'bool'>
```

Zauważ że inny typ związany jest z liczbami całkowitymi, inny z rzeczywistymi, a jeszcze inny z wartościami logicznymi (`True`/`False`). Zauważ także, że zmienna może zmienić swój typ.

1.1.1 Typowanie w Pythonie a w innych językach 😳

Typowanie, czyli określanie typu zmiennej, w Pythonie można porównać do typowania w współczesnym C++ z użyciem słowa kluczowego auto. Python:

- określa zmiennej w momencie napotkania jej deklaracji na podstawie wartości do niej przypisywanej (tak samo jak C++ robi dla zmiennych auto)

```
#include <stdio.h>
int main() {
  auto a = 1;
  printf("%d", a);
}
```

- nie pozwala odwołać się do zmiennej nie zadeklarowanej (np. PHP pozwala, generując jedynie ”Notice”)

```
<?php
$a = $b +1;
echo $a, $b;
?>
```

- pozwala na zmianę typu zmiennej w trakcie działania (C++ nie pozwala nawet z typem auto)

```
a = "abc"
print(a, type(a))
a = 1
print(a, type(a))
```

1.1.2 Wielkość zmiennej typu int

Python nie posiada wbudowanego ograniczania wielkości liczb całkowitych, jednak wielkość wartości przechowywanej w tym typie może mieć wpływ na rozmiar zmiennej.

```
x = 1
print(x, type(x), x.__sizeof__())
x = 12**10
print(x, type(x), x.__sizeof__())
x = 12**20
print(x, type(x), x.__sizeof__())
x = 13
print(x, type(x), x.__sizeof__())
```

```
1 <class 'int'> 28
61917364224 <class 'int'> 32
383375992447475122176 <class 'int'> 36
13 <class 'int'> 28
```

1.2 Zmienna, obiekt i referencja

W Pythonie każda zmienna jest nazwą wskazującą na jakiś obiekt w pamięci. Podobnie każdy element listy czy słownika wskazuje na jakiś obiekt[1]. Na jeden obiekt może wskazywać wiele zmiennych i/lub elementów innych obiektów (takich jak listy czy słowniki). Jeżeli zmienna nie ma na co wskazywać (np. został do niej przypisany wynik funkcji, która nie zwraca wartości) wskazuje na obiekt `None` (typu `NoneType`). Zatem na wszystkie zmienne pythonowe możemy patrzeć jak na referencje do obiektów istniejących gdzieś w pamięci.

Do uzyskania identyfikatora obiektu związanego z daną nazwą, lub elementem innego obiektu służy funkcja `id` (w przypadku standardowej implementacji Pythona jest to po prostu adres w pamięci).

1.2.1 Usuwanie i czas życia zmiennych

Instrukcja **`del`**, której używaliśmy już do usuwania elementów z listy lub słownika może być wykorzystana także do usuwania innych zmiennych. Należy jednak pamiętać iż w Pythonie usunięcie zmiennej nie wiąże się z natychmiastowym zwolnieniem zajmowanej przez nią pamięci z kilku powodów:

- na pojedynczy obiekt może wskazywać kilka zmiennych
- to Python decyduje o tym kiedy zwalniać / ponownie użyć pamięć pozostałą po obiektach na które nie wskazuje już żadna nazwa

1.2.2 Kopiowanie obiektów

Python w momencie przypisania wartości jednej zmiennej do innej nie tworzy kopii obiektu na który wskazuje zmienna, zamiast tego przypisuje referencję do istniejącego obiektu. Jest to szczególnie zauważalne w obiektach, które mogą być wewnętrznie modyfikowalne (takich jak listy czy słowniki)[2]:

1. Zasadniczo wszystkie definiowane przez nas zmienne czy funkcje są elementem słownika związanego z danym kontekstem. Do słowników tych można uzyskać dostęp poprzez funkcje `globals()` (słownik zawierający elementy zdeklarowane w kontekście globalnym) i `locals()` (słownik zawierający elementy zadeklarowane w kontekście lokalnym).
2. Zauważ że jedyną możliwością modyfikacji liczby czy napisu jest przypisanie wartości wyrażenia do zmiennej, a dla list czy słowników możemy je modyfikować bez operacji przypisania całej listy czy słownika do nowej czy tej samej zmiennej. Jest to podział na typy "immutable" i "mutable" - te pierwsze nie są wewnętrznie modyfikowalne (każda modyfikacja odbywa się przez przypisanie obiektu do zmiennej, w wyniku którego pod zmienną może zostać podpięty nowy obiekt).

```
a = [1, 2, 3]
b = a
print(a, b, "\n", hex(id(a)), hex(id(b)))
a[1] = 0
print(a, b, "\n", hex(id(a)), hex(id(b)))
del a
print(b,    "\n", hex(id(b)))
```

```
[1, 2, 3] [1, 2, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3] [1, 0, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3]
 0x7f50d76b2bc8
```

Jak widać a i b posiadają taki sam identyfikator obiektu zwracany przez funkcję id, modyfikacja a[1] wpłynęła na zawartość b, natomiast usunięcie a nie ma wpływu na b (usunęliśmy tylko jedną z dwóch referencji na wspólny obiekt). Jeżeli chcemy uzyskać kopię listy lub słownika musimy skorzystać z metody copy() odpowiedniego obiektu:

```
a = [1, 2, 3]
b = a.copy()
b[1] = "X"
print(a, b, "\n", hex(id(a)), hex(id(b)))
```

```
[1, 2, 3] [1, 'X', 3]
 0x7f50d76b2bc8 0x7f50d57a7088
```

Zauważ że tak utworzone b ma inny identyfikator obiektu niż a. Należy mieć także na uwadze że nawet argumenty funkcji przekazywane są jako referencje na obiekty a nie kopie obiektów, natomiast dopiero operacja przypisania nowej wartości do zmiennej związanej z argumentem powoduje że zaczyna ona wskazywać na nowo utworzony (w wyniku wyrażenia po prawej stronie znaku równości) obiekt.

1.2.3 Dla jeszcze bardziej dociekliwych 😳

Osobom jeszcze bardziej dociekliwym w temacie wnętrzności Pythona możemy polecić lekturę artykułu omawiającego te zagadnienia http://www.rwdev.eu/articles/objectthinking oraz samodzielne eksperymenty.

1.3 Obsługa błędów

Wcześniej spotkaliśmy się już z komunikatem błędu. Błędy mogą wynikać z błędów składniowych w programie ale również nie przewidzianych zdarzeń w trakcie jego pracy. Warto mieć na uwadze iż wszystkie błędy w Pythonie mają postać wyjątków które mogą zostać obsłużone blokiem **try**/**except**.

```
try:
  a = 5 / 0
except ZeroDivisionError:
  print("dzielenie przez zero")
except:
  print("inny błąd")
```

Przy obsłudze błędów może przydać się instrukcja pusta **pass**, która w tym przypadku pozwala na zignorowanie obsługi danego błędu.

```
try:
  slownik["a"] += 1
except:
  pass
```

Powyższy kod zwiększy wartość związaną z kluczem "a" w słowniku slownik, jednak gdy napotka błąd (np. słownik nie zawiera klucza "a") zignoruje go.

Możemy także generować wyjątki z naszego kodu, służy do tego instrukcja **raise**, której należy przekazać obiektem dziedziczącym po **BaseException** np:

```
raise BaseException("jakiś błąd")
```

1.4 Kod binarny

Jak wiemy liczby możemy zapisywać w różnych systemach liczbowych i jednym z nich jest system dwójkowy, nazywany też binarnym. Taka reprezentacja liczb jest podstawą działania elektroniki cyfrowej w tym współczesnych komputerów.

Napis przedstawiający liczbę w reprezentacji dwójkowej w Pythonie można z pomocą funkcji `bin`. Funkcja ta niestety nie pozwala wymusić długości wypisywanej liczby (co jest bardzo przydatne jeżeli chcemy operować na poszczególnych bitach) a dodatkowo liczby ujemne wypisuje ze znakiem minus i reprezentacją liczby dodatniej (czyli zasadniczo w kodzie znak moduł) a nie rzeczywiście stosowanym do zapisu takich liczb na zdecydowanej większości architektur kodzie uzupełnień do dwóch. Dlatego na potrzeby przykładów w tym rozdziale będziemy używać własnej funkcji zwracającej binarną reprezentację liczb 8bitowych (czyli 1 bajta):

```
def bin8(x):
        return "0b{0:08b}".format(x & 0xff)
```

Liczby dodatnie w systemie binarny zapisuje się praktycznie zawsze w postaci NKB. Zapis taki jest analogiczny do zapisu dziesiętnego stosowanego na co dzień, z tym że kolejne cyfry liczby mają wagę 2^n a nie 10^n (gdzie n jest numerem cyfry, zaczynającym się od zera dla skrajnie prawej).

$$a_n a_{n-1} ... a_1 a_0 \leftrightarrow a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Liczby ujemne mogą być zapisywane na różne sposoby. Wspomniany kod moduł-znak polega na zapisie modułu liczby w postaci NKB oraz umieszczenia flagi znaku w najstarszym bicie (0 – liczba dodatnia, 1 – ujemna). Najczęściej stosowany jest jednak kod uzupełnień do dwóch (określany jako U2) przypominający NKB tyle że najstarszy n-ty bit wchodzi z wagą $-(2^n)$ a nie 2^n:

$$a_n a_{n-1} ... a_1 a_0 \leftrightarrow -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

```
print(bin8(3),  bin(3))
print(bin8(-3), bin(-3))
```

```
0b00000011 0b11
0b11111101 -0b11
```

Możemy sprawdzić czy bin8 rzeczywiście wypisało reprezentację -3 w kodzie U2 wykonując proste obliczenie:

```
-2**7 + 2**6 + 2**5 + 2**4 + 2**3 + 2**2 + 0*(2**1) + 2**0
```

```
-3
```

1.4.1 Operacje bitowe

Python, jak wiele innych języków, pozwala wykonywać operacje boolowskie nie tylko na wartościach reprezentujących pradwę i fałsz, ale także na odpowiadajacych sobie bitach dwóch liczb. Operację bitowego AND zapisujemy z pomocą `&`, OR z pomocą `|`, XOR z pomocą `^`, a NOT z pomocą `~`:

```
print(bin8( 0b11001010 & 0b10101110 ))
print(bin8( 0b11001010 | 0b10101110 ))
print(bin8( 0b11001010 ^ 0b10101110 ))
print(bin8( ~0b11001010 ))
```

```
0b10001010
0b11101110
0b01100100
0b00110101
```

Jak widzimy w pokazanym przykładzie operacje te są wykonywane na każdym z bitów liczby niezależnie czyli n-ty bit wyniku bitowego AND to n-ty bit pierwszej liczby AND n-ty bit drugiej liczby, itd.

```
print(bin8( 0b11001010 << 3 ))
print(bin8( 0b11001010 >> 3 ))
```

```
0b01010000
0b00011001
```

Dostępne są także operacje przesunięcia bitów w ramach liczby w lewo lub prawo (brakujące bity uzupełniane są zerami, a bity wystające poza długość liczby binarnej są obcinane[3]). Operacje te odpowiadają mnożeniu i dzieleniu całkowitemu przez 2^x, gdzie x to ilość bitów do przesuniecia podawana po prawej stronie operatora przesuniecia w postaci << lub >>.

Operacje takie są przydatne do sprawdzania bądź ustawiania wartości poszczególnych bitów. Są to operacje dość niskopoziomowe i nie często stosowane w Pythonie, ale wiedza o nich przyda nam się w niedalekiej przyszłości.

1.5 Słowniki

Kolejnym użytecznym typem zmiennych w Pythonie są słowniki (zwane niekiedy *mapami* lub *tablicami asocjacyjnymi*). Podobnie jak listy służą do do przechowywania innych zmiennych. W odróżnieniu jednak od list w słownikach przechowywane są pary klucz - wartość, gdzie unikalny klucz służy do identyfikowania wartości. Zwróć uwagę na analogię z normalnymi słownikami klucz to słowo które wyszukujemy, a wartość to jego opis.

```
slownik = { "bd" : "xx", 5: True, "a" : 11 }
for klucz in slownik:
    print (klucz, "=>", slownik[klucz])
```

```
a => 11
bd => xx
5 => True
```

Zauważ że zarówno klucz, jak i wartość mogą być dowolnego typu oraz że słownik nie zachowuje kolejności dodawania elementów.

Możliwe jest także sprawdzanie istnienia jakiegoś elementu w słowniku, usuwanie, dodawanie i zmienianie elementów słowniku, itd (zwróć także uwagę na inną metodę wypisywania słownika - poprzednio iterowaliśmy po kluczach, teraz po liście par klucz-wartość):

```
if "bd" in slownik:
    print ("jest element o kluczu 'bd'")
    del slownik['bd']
slownik[15] = True
slownik["a"] = "yy"
for k,v in m.items():
    print (k, "=>", v)
```

```
jest element o kluczu 'bd'
a => yy
15 => True
```

1.5.1 Sortowanie słownika

Jak już wspomnieliśmy słownik nie zachowuje porządku elementów. Jeżeli chcemy uzyskać posortowaną listę kluczy, wartości lub par klucz-wartość z słownika możemy skorzystać z funkcji `sorted()`. W przypadku

3. W przypadku Pythona liczby całkowite nie mają maksymalnej wielkości, a obcinanie przy przesuwaniu w lewo realizuje nasza funkcja wypisująca bin8.

par wywołanie będzie wyglądać następująco:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
lista = sorted( mapa.items() )
print(lista)
```

```
[('5', 3), ('a', 101), ('bd', 20)]
```

Zwróć uwagę, iż użyliśmy tej samej metody `items()`, z której korzystaliśmy do iterowania po parach klucz-wartość (dla listy samych kluczy lub wartości należy użyć w tym miejscu innej metody klasy `dict`). Zapewne zauważyłeś że sortowanie zostało przeprowadzone w oparciu o klucze, co jednak jeżeli chcielibyśmy posortować taką listę w oparciu o wartości? W takim przypadku możemy skorzystać z opcjonalnego argumentu funkcji `sorted()` o nazwie `key`, który przyjmuje funkcję mającą za zadanie na podstawie otrzymanego elementu listy (w tym wypadku pary klucz - wartość) zwrócić klucz sortowania:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
def k(x):
    return x[1]
lista = sorted( mapa.items(), key=k )
print(lista)
```

```
[('5', 3), ('bd', 20), ('a', 101)]
```

1.6 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.

1.7 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
  @staticmethod
  def info():
    print("INFO")
  # konstruktor (z jednym argumentem)
  def __init__(self, x = 1):
    print("konstruktor", self.a , self.d)
    # i kolejny sposób na utworzenie pola składowego klasy
    self.b = 13 * x
```

Warto zauważyć jawny argument metod składowych klasy w postaci obiektu tej klasy. W innych językach programowania ten argument także występuje, ale często jest ukryty przed programistą - nie podajemy do ani w definicji metody, ani przy odwołaniach do pól klasy w metodzie (np. w C++).

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[4]:

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

```
INFO
80
7
```

Obiekty klas są obiektami modyfikowalnymi, zatem jak wiemy zwykłe przypisanie tworzy tylko inną referencję na ten sam obiekt. Celem utworzenia kopii naszego obiektu możemy zaimplementować własną metodę copy lub skorzystać z funckji copy dostarczanej przez moduł copy.

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

1.8 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):
        yield a
        a, b = b, a + b

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:
      yield a
      a, b = b, a + b
```

1.9 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.

1.9.1 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[5]. 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.

1.9.2 Wczytywanie tekstu z pliku

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

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()
```

Po otwartym pliku możemy się przemieszczać metodą `seek`, na przykład `plik.seek(0)` przesunie punkt odczytu na początek pliku i umożliwi jego ponowne przeczytanie.

1.9.3 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)
```

5. 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.

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

pid = os.fork()
if pid == 0:
        print("potomek: mój pid to", os.getpid())
else:
        print("rodzic: pid potomka to", pid)
```

```
pid to: 8763
rodzic: pid potomka to: 8764
potomek: mój pid to 8764
```

Przykład możemy trochę rozbudować używając funkcji sleep aby zaobserwować współistnienie tych dwóch procesów oraz funkcji signal do zakończenia procesu potomnego przez rodzica:

```python
import os, time, signal

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

pid = os.fork()
if pid == 0:
        print("potomek: mój pid to", os.getpid())
        time.sleep(4)
        print("potomek 1")
        time.sleep(7)
        print("potomek 2")
else:
        print("rodzic: pid potomka to", pid)
        time.sleep(5)
        print("rodzic 1")
        time.sleep(4)
        print("zabijam potomka")
        os.kill(pid, signal.SIGTERM)
        time.sleep(5)
        print("rodzic 2")
```

```
pid to: 5295
rodzic: pid potomka to 5301
potomek: mój pid to 5301
potomek 1
rodzic 1
zabijam potomka
rodzic 2
```

2.2 wywołanie zewnętrznej komendy

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

```python
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*:

```python
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*[6]. Jednak, jako że w ramach tego kursu nie będziemy zajmować się programowaniem równoległym jako takim, to modułów tych nie omówimy w 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.

Reguły DRY i KISS

„Don't Repeat Yourself" (*nie powtarzaj się*) jest jedną z dwóch głównych reguł programistycznych (ale ma także pewne zastosowania w innych dziedzinach techniki). Zaleca ona unikanie potarzania tych samych czynności, czy też tworzenia takich samych, a nawet analogicznych, podobnych fragmentów kodu.
Narzędziami ułatwiającymi realizację tego celu są m.in.:

- systemy i skrypty służące automatyzacji różnego rodzaju czynności (takich jak np. kompilacja, instalacja, aktualizacja, monitoring działania) – zarówno systemy takie jak make, cmake, doxygen ale również wszystkie drobne skrypty (np. shellowe czy pythonowe) tworzone w tym celu w codziennej pracy informatyka
- elementy składniowe (m.in. takie jak pętle i funkcje) oraz mechanizmy (np. polimorfizm) dostępne w językach programowania pozwalające na eliminację powtórzeń kodu
- biblioteki, moduły, itp pozwalające na współdzielenie tych samych rozwiązań, tego samego kodu, pomiędzy różnymi projektami
- elementy biblioteki systemowej pozwalające na wywoływanie innych programów (np. exec) i komunikację z nimi (np. poprzez strumienie wejścia/wyjścia)

Unikanie powtórzeń takiego samego lub (co często nawet gorsze) tylko nieznacznie zmienionego kodu jest też szczególnie istotne ze względu na łatwość utrzymania kodu – np. jakąś poprawkę wprowadza się tylko w odpowiednio sparametryzowanej funkcji, a nie kilkunastu podobnych (ale nie identycznych, ze względu na brak parametryzacji) fragmentach kodu.
W zastosowaniach nie programistycznych przejawia się często wydzielaniem modułów i dążeniem do ich powtarzalności, redukcji ilości ich typów (np. dzięki parametryzacji, czy konfigurowalności).

Drugą, nawet chyba ważniejszą, z tych dwóch reguł jest **„Keep It Simple, Stupid"** (niekiedy *Keep It Small and Simple*), którą można streścić jako *proste jest lepsze*. Reguła KISS jest bardziej ogólna (można nawet powiedzieć że wynika z niej reguła DRY), posiada dużo szersze pole zastosowań (także nie technicznych) i może być uważana za implementację *Brzytwy Ockhama* w inżynierii. Zaleca ona m.in.:

- tworzenie przejrzystych, czytelnych i prostych rozwiązań (zarówno pod względem samego projektu, koncepcji, jak też ich implementacji, wykonania)
- wybór rozwiązania prostszego spośród (równie) skutecznych rozwiązań jakiegoś problemu
- myślenie o łatwości późniejszego utrzymania i serwisu tworzonego rozwiązania (czy to kodu programu, czy urządzenia elektronicznego, a nawet budynku)

W duchu prostoty nakazywanej regułą KISS należy także starać się trzymać powszechnie stosowanych standardów (gdy tylko jest to możliwe i nie powoduje zbyt wielkiej komplikacji naszego projektu), zamiast każdorazowo tworzyć nowe, własne standardy, protokoły czy interfejsy.

Do tej pory korzystaliśmy z elementów standardowej biblioteki dostarczanej z Pythonem. W rozdziale

6. 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 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`.

Poniższe przykłady służą głównie zaprezentowaniu potencjału możliwego do uzyskania dzięki dostępnym bibliotekom, tego że są one dużym ułatwieniem dla programisty oraz pokazaniu kilku standardów związanych z zapisem danych.

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.

Argumenty linii poleceń

Tworzone przez nas skrypty pythonowe mogą przyjmować (tak jak różne inne poznane programy) argumenty (i opcje) w linii poleceń. Aby mieć dostęp do listy argumentów przekazanych w linii poleceń należy skorzystać z `sys.argv`:

```
import sys
print(sys.argv)
```

```
$ python3 /tmp/skrypt.py aa tt
['/tmp/skrypt.py', 'aa', 'tt']
```

Jak możemy zauważyć jest to standardowa pythonowa lista zawierająca w kolejnych swoich elementach nazwę z którą został wywołany nasz skrypt i kolejne przekazane do niego argumenty.

Moduł argparse dostarcza dość wygodny parser opcji w standardowej notacji (długie z dwoma myślnikami, krótkie z jednym, itd.):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument(
  '-v', "--verbose", action="store_true",
  help='opcja typu przełącznik'
)
parser.add_argument(
  'ARG', nargs='?',
  help='argument pozycyjny (opcjonalny)'
)
args = parser.parse_args()
print(args)
```

```
$ python3 /tmp/skrypt.py aa tt
usage: skrypt.py [-h] [-v] [ARG]
skrypt.py: error: unrecognized
 ↪  arguments: tt
$ python3 /tmp/skrypt.py -v aa
Namespace(ARG='aa', verbose=True)
```

To tylko mały przykład możliwości argparse, a szczegóły (jak zwykle) – w dokumentacji.

4 Wykład wideo

- *Python: typy i referencje* – https://www.youtube.com/watch?v=LiMPlGOX9e0
- *Python: operacje bitowe i wyjątki* – https://www.youtube.com/watch?v=fU3FwceTIM0
- *Python: słowniki i klasy* – https://www.youtube.com/watch?v=K0XdD6aygwo
- *Python: pliki i czekanie na dane* – https://www.youtube.com/watch?v=OhMh5yYZgBg
- *Python: procesy potomne (fork i exec)* – https://www.youtube.com/watch?v=Zy_T9lhhPCw
- *Python: biblioteki (xml, json, sql, ...)* – https://www.youtube.com/watch?v=kcHf4l66QbA

5 Literatura dodatkowa

- *The Python Tutorial* (https://docs.python.org/3/tutorial/) - oficjalny Tutorial Pythona.
- *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).
- *Vademecum informatyki praktycznej* (http://vip.opcode.eu.org/) - zbiór materiałów na temat elektroniki i programowania, zawierający także dość rozbudowaną listę literatury dodatkowej.
- *Linux i Python w Elektronicznej Sieci* (https://ciekawi.icm.edu.pl/lpes) - strona domowa kursu *LPES*, zawierająca nagrania i skrypty do innych wykładów, skrypty ćwiczeniowe, itd.
- *OpCode.eu.org* (http://vip.opcode.eu.org/) - strona internetowa autora kursu *LPES*, zawierająca różne materiały z szeroko rozumianej inżynierii komputerowej i elektronicznej (część materiałów pokrywa się z zawartością skryptów z tego kursu, ale nie wszystkie)

6 Zadania

Zadanie 6.0.1

Zapoznaj się z dokumentacją klasy odpowiedzialnej za napisy (`str`), zwróć szczególną uwagę na metody `split`, `find`, `replace`. Korzystając z metod klasy `str` napisz funkcję `parse` która dla napisu będącego jej argumentem wykona zamianę wszystkich ciągów "XY" na spację oraz dokona rozbicia napisu złożonego z pól rozdzielanych dwukropkiem na listę napisów odpowiadających poszczególnym polom. Funkcja powinna działać w następujący sposób:

```
 > l = parse("Ala:maXYkota:i inne:zwierzeta")
 > print(l)
['Ala', 'ma kota', 'i inne', 'zwierzeta']
```

Zadanie 6.0.2

Napisz funkcję `zlicz` która dla podanej listy policzy powtórzenia jej elementów. Przykład użycia:

```
 > zlicz(["AX", "B", "AX"])
AX wystepuje 2 razy
B wystepuje 1 razy
```

Wskazówka: Użyj słownika, w którym element będzie stanowił klucz, a krotność jego wystąpień wartość. Możesz użyć metody `get()` do pobierania wartości z słownika, jeżeli w nim jest lub wartości domyślnej w przeciwnym wypadku - szczegóły zobacz w dokumentacji

Zadanie 6.0.3

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 6.0.4

Zastanów się czy konstrukcję if/elif w funkcji `dzialanie()` z rozdziału 1.6 można by zastąpić słownikiem, jak to ewentualnie zrobić i jakie mogłoby mieć to zalety bądź wady?

Zadanie 6.0.5

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 6.0.6

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 -Af` w taki sposób aby potomek odebrał do zmiennej jej standardowe wyjście i wypisał je na ekran.

Zadanie 6.0.7

Napisz funkcję `zapisz` 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.
Na przykład dla wywołania `zapisz({"a": "qwe", "d": "123"}, "xx")` funkcja powinna utworzyć plik z zawartością:

```
a	qwe
d	123
```

7 Rozwiązania

Poniżej zamieszczone są przykładowe rozwiązania „głównych" zadań z tego skryptu wraz z komentarzami. Wiemy że zajrzenie do nich już przy pierwszej trudności jest kuszące, mimo to rekomendujemy przynajmniej podjąć uczciwą, co najmniej kilkunastominutową na każde z zadań, próbę rozwiązania tych zadania bez zaglądania do odpowiedzi.

Pamiętaj!: Samodzielne rozwiązanie problemu (wraz z wszystkimi trudnościami po drodze i popełnionymi błędami) jest dużo bardziej kształcące od nawet wielokrotnego przepisania gotowego rozwiązania, jednak nawet jednokrotne przepisanie rozwiązania jest bardziej kształcące od wielokrotnego przekopiowania go.

Rozwiązanie zadania 6.0.1

```
def parse(t):
    t = t.replace("XY", " ")
    return t.split(":")
```

Zwróć uwagę że:

- używamy metody `replace` na oryginalnym napisie celem zastąpienie XY spacją, należy zauważyć że metoda ta nie modyfikuje oryginalnego napisu tylko zwraca nowy (zmieniony) napis, dlatego zapisujemy jej wynik do zmiennej, celem dalszego przetwarzania
- używamy metody split z określeniem separatora w jej argumencie, zwraca ona listę powstałą z podziału napisu przy pomocy wskazanego separatora
- listę tą zwracamy z funkcji za pomocą return

Rozwiązanie zadania 6.0.2

```
def zlicz(l):
    s = {}
    for e in l:
        s[e] = s.get(e, 0) + 1
    for k in s:
        print (str(k) + " wystepuje " + str(s[k]) + " razy")
```

Zwróć uwagę że:

- wykorzystujemy słownik s do trzymania mapy element - ilosć powtórzeń
- przed pętlą zliczającą inicjujemy s jako pusty słownik
- w pętli zliczającej (iterującej po liście) używamy metody get słownika, aby pobrać wartość odpowiadającą danemu kluczowi lub zero gdy takiego klucza nie było, zamiast tej metody mogliśmy użyć konstrukcji `if e in s:` do rozróżnienia przypadku pierwszego i kolejnego wystąpienie elementu e.
- po pętli zliczającej mamy osobną pętlę iterującą po słowniku celem wypisania ilości wystąpień

Rozwiązanie zadania 6.0.3

```
def wykonaj(lista, funkcja):
    for x in lista:
        funkcja(x)
```

Zwróć uwagę że funkcję przekazujemy do zmiennej tak samo jak dowolny inny argument (zmienną). Użycie funkcji przechowywanej w zmiennej polega na wywołaniu tej zmiennej z nawiasami okrągłymi i ewentualnymi argumentami tej funkcji.

Rozwiązanie zadania 6.0.4

Możemy zdefiniować słownik, w którym kluczem jest nazwa działania a wartością funkcja je realizująca. Zaleta takiego podejścia jest łatwe rozszerzania takiego kodu o nowe działania (poprzez wstawienie kolejnej pary do słownika, co może dziać się w trakcie pracy programu).

Rozwiązanie zadania 6.0.5

```
import sys, os, select

def czytaj(timeout):
    bufor = ""
    while True:
        rdfd, _, _ = select.select([sys.stdin], [], [], timeout)

        if not rdfd:
            return bufor

        for fd in rdfd:
            bufor += os.read(fd.fileno(), 1024).decode()

czytaj(13)
```

Zwróć uwagę że:

- funkcja w nieskończonej pętli wykonuje `select` z ustawioną wartością timeout
- w oparciu o wynik działania select funkcja rozróżnia przypadek timeoutu (rdfd jest `None`, czyli `not` rdfd jest prawdą) i zwraca w tej sytuacji wczytane wcześniej dane
- jeżeli nie było timeoutu, a pojawiły się jakieś dane (rdfd nie jest `None`) to funkcja wczytuje je z użyciem funkcji read
- jako że funkcja read wymaga określenia jakiegoś skończonej wielkości bufora, to do wczytywania danych użyta jest pętla for, co zapewnia wczytanie wszystkich dostępnych w danym momencie danych
- na wczytanych danych użyta jest metoda decode w celu zamiany ciągu bitowego na napis i tak przekonwertowane dane dodawane są do bufora napisowego (o dynamicznie dostosowywanej przez Pythona długości)

Rozwiązanie zadania 6.0.6

```
import os, subprocess

pid = os.fork()

if pid == 0:
    res = subprocess.run(["ps", "-Af"], stdout=subprocess.PIPE)
    print("Standardowe wyjście z komendy to: " + res.stdout.decode())
else:
    print("Mój PID to ", os.getpid(), "PID mojego potomka to:", pid)
```

Zwróć uwagę że:

- korzystamy z funkcji fork do rozdzielania procesu na dwa (rodzica i potomka)

- każdy z tych procesów zaczyna wykonywanie od wyjścia z funkcji fork, przy czym różni się w nich wartość, którą ta funkcja zwróciła
- wartości tej używamy do rozróżnienia rodzica od potomka
- w rodzicu (który otrzymał niezerową wartość, będącą numerem PID utworzonego potomka) wypisujemy stosowny komunikat na temat numerów PID
- w potomku używamy `subprocess.run` do uruchomienia wskazanego polecenia w kolejnym potomku i przechwycenia jego wyjścia (opcjonalny argument `stdout=subprocess.PIPE`)
- po zakończeniu działania subprocess wypisujemy dane otrzymane na standardowym wyjściu uruchomionego polecenia
- dane te w ogólności są danymi binarnymi i Python używa typu ciągu bajtowego do ich przechowywania, my wiemy że ps wypisuje tekst na standardowym wyjściu zatem korzystamy z metody `decode` do jego konwersji na napis

Rozwiązanie zadania 6.0.7

```
def zapisz(slownik, nazwa):
    plik = open(nazwa, 'wt', encoding='utf8')
    for klucz in slownik:
        plik.write(klucz + "\t" + slownik[klucz] +"\n")
    plik.close()
```

Zwróć uwagę że:

- funkcja użyta open z argumentami wskazującymi otwarcie pliku tekstowego w trybie do zapisu, w tym przykładzie podaliśmy także kodowanie, ale jest to zbędne gdyż podany utf8 byłby i tak użyty domyślnie
- następnie w ramach pętli przechodzącej po przekazanym do funkcji słowniku (dokładnie po kluczach w tym słowniku) wywołujemy funkcję write i podajemy do niej odpowiednio przygotowany napis (jako że plik otworzyliśmy w trybie tekstowym, write przyjmuje napisy)
- po wpisaniu wszystkich danych dokonujemy zamknięcia pliku (bez wykonania tej operacji część danych mogłaby nie być fizycznie zapisana w pliku w momencie kończenia funkcji)