Laboratorium programistyczne: wprowadzenie do programowania w Pythonie

Projekt „Matematyka dla ciekawych świata”
Robert Paciorek, Łukasz Mazurek

2018-04-05

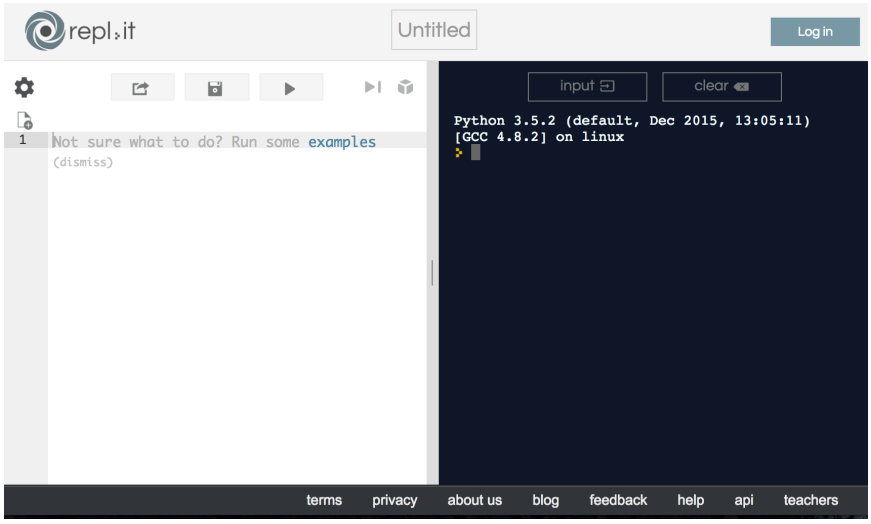
1 Praca z Pythonem

Na zajęciach będziemy programować w języku Python w wersji 3. Pythona można używać m.in. na jeden z dwóch sposobów:

Zainstalowany interpreter Pythona. Interpreter Pythona jest zainstalowany na komputerach w laboratorium. Możecie go również bezpłatnie pobrać ze strony https://www.python.org/downloads/ i zainstalować na swoich domowych komputerach. W przypadku systemów z rodziny Linux Python na ogół jest dostępny w repozytoriach danej dystrybucji i można go zainstalować poprzez menagera pakietów (np. `apt-get install python3`). Można także zainstalować i używać wygodniejszy w pracy interaktywnej interpreter `ipython3`.

Zwróćcie uwagę, aby zainstalować i uruchamiać wersję o numerze rozpoczynającym się od 3 (np. 3.6.0), a nie starszą, ale wciąż używaną wersję 2. Wersje te różnią się tak znacząco, że programy, które będziemy pisać na zajęciach nie będą działać w starszej, drugiej wersji Pythona.

Interpreter Pythona online. Istnieje wiele interpreterów Pythona online. Na zajęciach będziemy korzystali z interpretera znajdującego się na stronie http://repl.it . Interpretera tego będziecie mogli również używać na swoich domowych komputerach bez instalacji żadnego dodatkowego oprogramowania — wystarczy dowolna przeglądarka i podłączenie do internetu.



Rysunek 1: Widok interpretera Pythona dostępnego na stronie http://repl.it

Innymi zasługującymi na uwagę interpreterami online są: http://www.tutorialspoint.com/execute_python_online.php i http://ideone.com

1.1 Praca z interpreterem `repl.it`

Uruchom przeglądarkę internetową i wejdź na stronę http://repl.it. W polu *Search for a language* na środku ekranu wybierz język *Python3*. Pojawi się okienko do zalogowania — można kontynuować bez logowania po kliknięciu *continue as Anonymous*. Powinieneś zobaczyć dwa okienka: miejsce na kod (białe, z lewej) i *konsolę* (czarne, z prawej, zwane również *terminalem*), tak jak na rysunku 1.

2 Praca z konsolą interaktywną

Pierwszym sposobem pracy z Pythonem jest praca w interaktywnej konsoli, czyli w przypadku `repl.it` praca w prawym okienku. W konsoli tej początkowo wypisane są pewne informacje (m. in. używana wersja Pythona) oraz znak zachęty `>`. Interpreter oczekuje, iż po tym znaku wpiszemy polecenie i naciśniemy Enter. Wynik polecenia zostanie wypisany w kolejnym wierszu.

Najprostszym sposobem użycia konsoli Pythona jest użycie jej jako kalkulatora — wpisujemy działanie do obliczenia, naciskamy Enter i w kolejnym wierszu otrzymujemy wynik działania. Przykład użycia konsoli Pythona jako kalkulatora znajduje się poniżej:

```
 > 2 + 2 * 2
=> 6
 > (2 + 2) * 2
=> 8
 > 2 ** 7
=> 128
 > 47 / 10
=> 4.7
 > 47 // 10
=> 4
 > 47 % 10
=> 7
```

W powyższym przykładzie:

- Znak `**` oznacza podnoszenie do potęgi.
- Znak `/` oznacza dzieleie.
- Znak `//` oznacza dzielenie całkowite.
- Znak `%` oznacza branie reszty z dzielenia.

Podobnie jak w kalkulatorze możemy korzystać z *pamięci*, w Pythonie możemy zapisywać wartości w *zmiennych*:

```
 > x = 3
 > y = 4
 > x
=> 3
 > x**2 + y**2
=> 25
```

W pierwszych dwóch linijkach następuje *przypisanie* wartości 3 do zmiennej `x` oraz wartości 4 do zmiennej `y`. Od tej pory możemy korzystać z tych zmiennych, np. do obliczenia wartości wyrażenia $(x^2 + y^2)$.

Zadanie 1 *Korzystając z Pythona jak z kalkulatora, znajdź wszystkie dziesięciocyfrowe potęgi dwójki.*

2.1 Pętla for

Załóżmy, że chcemy obliczyć kwadraty wszystkich liczb od 1 do 4. Zgodnie z dotychczasową wiedzą, w tym celu musimy wykonać 4 działania:

```
 > 1 * 1
=> 1
 > 2 * 2
=> 4
```

```
 > 3 * 3
=> 9
 > 4 * 4
=> 16
```

Widzimy jednak, że te działania są bardzo podobne i chciałoby się je wykonać „za jednym zamachem”. Do wykonywania wielokrotnie tego samego (lub podobnego) kodu służą pętle. Najprostszym rodzajem pętli jest pętla **for**, która dla danej *listy* i operacji do wykonania wykonuje tę operację po kolei na każdym elemencie listy.

Do wykonania powyższego zadania służy pętla **for** w następującej postaci:

```
 > for x in [1, 2, 3, 4]:
..     print(x * x)
..
1
4
9
16
```

Spróbuj przepisać tę pętlę do konsoli interpretera i uruchomić. Zwróć uwagę na kilka rzeczy:

- Na końcu pierwszej linijki jest dwukropek.
- Gdy naciśniemy Enter po zakończeniu pierwszej linijki, znak zachęty zmieni się z `>` na `..`, co oznacza to, że jesteśmy w trakcie pisania polecenia wielolinijkowego.
- Druga linijka musi być *wcięta*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji. Jeśli zapomnimy o wcięciu, interpreter zgłosi błąd i całą komendę wielolinijkową będziemy musieli pisać od początku (można pomóc sobie, naciskając strzałkę w górę i przeglądając stare komendy).
- Po wpisaniu drugiej linijki i naciśnięciu Entera pojawi się kolejny znak zachęty `..`, co oznacza, że interpreter czeka na ciąg dalszy polecenia wielolinijkowego. Jeśli nie chcemy pisać dalszego ciągu, w tym momencie musimy jeszcze raz nacisnąć Enter.
- Do wypisania wyniku używamy funkcji **print**`()`. Nie korzystaliśmy z niej wcześniej, ponieważ bazowaliśmy na domyślnym zachowaniu interpretera przy pracy interaktywnej powodującym wypisywanie na konsolę wyniku nie zapisywanego do zmiennej. Jednak kiedy tworzymy program powinniśmy w jawny sposób określać co chcemy aby zostało wypisane na konsolę np. przy pomocy funkcji **print**`()`

3 Pisanie i uruchamianie kodu programu

Do tej pory korzystaliśmy z Pythona używając interaktywnej konsoli. Jest to całkiem wygodne narzędzie, jeśli wykonujemy tylko jednolinijkowe polecenia, jednak pisanie dłuższych fragmentów kodu w tej konsoli staje się już bardzo niewygodne. Drugą metodą korzystania z Pythona jest pisanie kodu programu (skryptu) w pliku tekstowym (w przypadku `repl.it` w lewym okienku) i uruchamianie tego kodu w konsoli (w prawym okienku).

Spróbuj teraz napisać w lewym okienku kod tej samej pętli for (pamiętając o dwukropku na końcu pierwszej i wcięciu drugiej linijki):

```
for x in [1, 2, 3, 4]:
    print(x * x)
```

i wcisnąć przycisk „run” znajdujący się na górze tego okienka. Po chwili w konsoli po prawej powinien pojawić się wynik działania naszego programu:

```
1
4
9
16
```

Od tej pory właśnie w taki sposób będziemy pisać i uruchamiać wszystkie nasze programy. Ilekroć w niniejszych materiałach pojawią się dwie ramki, jedna obok drugiej, w lewej ramce znajdował się będzie kod programu, a w prawej efekt jego działania wyświetlony w konsoli:

```
for x in [1, 2, 3, 4]:
    print(x * x)
```

```
1
4
9
16
```

Powyższa pętla wypisała każdą liczbę w osobnej linijce. Dzieje się tak, ponieważ polecenie **print**`(...)` domyślnie przechodzi do następnej linii po każdym wywołaniu. Można jednak zmienić to zachowanie, dodając wewnątrz **print**`(...)` po przecinku końcówkę `end = X`, gdzie `X` to otoczony apostrofami ciąg znaków, który chcemy wypisywać zamiast przejścia do nowej linii.

Przykłady:

```
for x in [1, 2, 3, 4, 5]:
    print(x * x, end = ' ')
```

```
1 4 9 16 25
```

```
for x in [1, 2, 3, 4, 5]:
    print(x * x, end = '')
```

```
1491625
```

```
for x in [1, 2, 3, 4, 5]:
    print(x * x, end = ' , ')
```

```
1 , 4 , 9 , 16 , 25 ,
```

3.1 Lista kolejnych liczb naturalnych

Często potrzebujemy, aby pętla przeszła po liście kilku kolejnych liczb naturalnych. W tym celu możemy oczywiście podać wprost kolejne elementy listy (tak jak w powyższym przykładzie), jednak istnieje wygodniejsze rozwiązanie, mianowicie polecenie **range**`()`:

```
for x in range(7):
    print(x, end = ', ')
```

```
0, 1, 2, 3, 4, 5, 6,
```

```
for x in range(5, 10):
    print(x, end = ', ')
```

```
5, 6, 7, 8, 9,
```

```
for x in range(10, 20, 3):
    print(x, end = ', ')
```

```
10, 13, 16, 19,
```

Na powyższych przykładach widzimy, że polecenie **range**`()` występuje w trzech wersjach:

- **range**`(kon)` generuje listę kolejnych liczb od 0 (**włącznie**) do `kon` (**wyłącznie**).
- **range**`(pocz, kon)` generuje listę kolejnych liczb od `pocz` (**włącznie**) do `kon` (**wyłącznie**).
- **range**`(pocz, kon, krok)` generuje listę liczb od `pocz` (**włącznie**) do `kon` (**wyłącznie**), przeskakując w każdym kroku o `krok`.

Do zapamiętania: ***Wszystkie przedziały w Pythonie są domknięte z lewej strony i otwarte z prawej strony, tzn. zawierają swój lewy koniec i nie zawierają swojego prawego końca.***

Zadanie 2 *Oblicz sumę* $1^2 + 2^2 + 3^2 + \ldots + 99^2 + 100^2$.

3.2 Instrukcja warunkowa if

Często chcemy, aby program zachowywał się w różny sposób w zależności od tego, czy jakiś warunek jest spełniony, czy nie. W Pythonie (jak w większości języków programowania) służy do tego instrukcja warunkowa **if**.

Przypuśćmy, że chcemy napisać pętlę, która dla kolejnych liczb od 1 do 5 sprawdzi czy jest parzysta, czy nie i wypisuje odpowiedni komunikat. Aby sprawdzić, czy liczba jest nieparzysta, możemy sprawdzić, czy reszta z dzielenia jej przez 2 jest różna 0. Zatem kod będzie wyglądał następująco:

```
for x in range(1, 6):
    if x % 2 != 0:
        print(x, '-- nieparzysta')
    else:
        print(x, '-- parzysta')
```

```
1 -- nieparzysta
2 -- parzysta
3 -- nieparzysta
4 -- parzysta
5 -- nieparzysta
```

Zwróć uwagę na następujące rzeczy:

- **if** to po polsku „jeśli”, **else** to po polsku „w przeciwnym przypadku”.
- Linijki rozpoczynające się od **if** i **else** (podobnie jak linijki rozpoczynające się np. od **for**) kończą się dwukropkiem.
- „Wnętrze” **if**-a i **else**-a (linijki 3 i 5) jest wcięte (podobnie jak wnętrze np. **for**-a).
- Linijka 3 zostanie wykonana, jeśli spełniony będzie warunek z linijki 2, czyli jeśli reszta z dzielenia liczby przez 2 będzie równa 0.
- Linijka 5 zostanie wykonana, jeśli warunek z linijki 2 nie będzie spełniony, czyli jeśli reszta z dzielenia liczby przez 2 nie będzie równa 0.

W powyższym przykładzie użyliśmy konstrukcji **if**/**else** do rozróżnienia pomiędzy dwoma przypadkami. Używając komendy **elif** (skrót od **else if**) możemy stworzyć bardziej skomplikowany kod do rozróżnienia pomiędzy kilkoma różnymi przypadkami:

```
for x in range(0, 5):
    if x < 1 or x == 4:
        print('mniejsze od 1 lub równe 4')
    elif x in [0,2,3]:
        print('0 2 lub 3')
    else:
        print('nic ciekawego')
```

```
mniejsze od 1 lub równe 4
nic ciekawego
0 2 lub 3
0 2 lub 3
mniejsze od 1 lub równe 4
```

Ten kod składa się z trzech bloków, które są wykonywane w zależności od spełnienia poszczególnych warunków: **if**, **elif**, **else**. Mamy dużą dowolność w konstruowaniu tego typu fragmentów kodu: bloków **elif** może być dowolnie wiele, blok **else** może występować jako ostatni blok, ale może też go nie być w ogóle. W powyższym przykładzie widzimy również, jak można łączyć warunki spójnikiem **or** („lub”) oraz **and** („i”) Pierwsze połączenie oznacza, że chcemy, aby spełniony był **co najmniej jeden** z wymienionych warunków, a drugie, że chcemy aby spełnione były **wszystkie** z wymienionych warunków.

Warunki można negować za pomocą słowa kluczowego **not** oraz grupować (celem wymuszenia kolejności działań) przy pomocy nawiasów okrągłych. Można korzystać m.in. z następujących operatorów porównań: `<` (mniejsze), `>` (większe), `<=` (mniejsze równe), `>=` (większe równe), `==` (równe), `!=` (nierówne)

Zwróć uwagę na środkowy warunek. Ma on postać „`if A in B:`”. Taki warunek sprawdza, czy wartość reprezentowana przez `A` jest elementem `B`. W naszym przykładzie sprawdzaliśmy, czy wartość zmiennej `x` występuje w podanej liście liczb, czyli czy jest 1, 2 lub 3.

Zauważ, że dla x wynoszącego 0 spełnione są dwa warunki (pierwszy i środkowy), w takim wypadku decydująca jest kolejność warunków i w konstrukcji **if**/**else** wykonany zostanie jedynie kod związany z pierwszym pasującym warunkiem.

3.3 Pętla while

Do tej pory korzystaliśmy z pętli **for**, która pozwala na iterowanie po liście elementów. Innym istotnym rodzajem pętli jest pętla **while**, która powoduje wykonywanie zawartego w niej kodu dopóki podany warunek jest spełniony.

```
a, b = 0, 1
while a <= 20:
    print(a, end=" ")
    a, b = b, a + b
```

```
0 1 1 2 3 5 8 13
```

Zwróć uwagę, że wewnątrz pętli **while** (tak samo jak innych konstrukcji używających wciętego bloku - takich jak **for**, czy **if**) może znajdować się więcej niż jedno polecenie. Trzeba tylko pamiętać, aby wszystkie były poprzedzone takim samym wcięciem.

3.4 Wielokrotne przypisanie

Zwróć uwagę w powyższym kodzie także na operację wielokrotnego przypisania postaci `a, b = x, y`. Dokonuje ona przypisania wartości x do a i y do b, przy wartości x i y obliczane są przed zmodyfikowaniem a i b. Pozwala to m.in. na zamianę wartości pomiędzy a i b bez stosowania zmiennej tymczasowej poprzez zapis: `a, b = b, a`. Podobnie możemy zapisywać przypisania większej ilości wartości do większej ilości zmiennych np: `a, b, c = 1, 5, 9`. Z notacji tej będziemy też często korzystać w dalszej części skryptu przy inicjalizacji zmiennych.

3.5 Definiowanie własnych funkcji

Wszystkie programy, które do tej pory pisaliśmy można podzielić na dwa rodzaje:

- ciąg jednolinijkowych poleceń, które wpisujemy bezpośrednio w konsoli Pythona (prawe, czarne okienko na stronie `repl.it`),
- pętle, które wygodniej pisać w pliku z kodem źródłowym (lewym, białym okienku na stronie `repl.it`), gdyż składają się z kilku linijek

Często będziemy chcieli wielokrotnie wykorzystać raz napisany fragment kodu. W tym celu będziemy tworzyć własne *funkcje*. Definicja funkcji ma następującą postać:

```
def <nazwa_funkcji>(<argumenty>):
  <pierwsze_polecenie>
  <drugie_polecenie>
  ...
```

Zwróć uwagę na podobieństwa definicji funkcji do pętli **for** oraz instrukcji warunkowej **if**:

- pierwsza linijka kończy się dwukropkiem,
- każda z kolejnych linijek jest *wcięta* względem pierwszej linijki.

Jako że definicje funkcji składają się zawsze z co najmniej kilku linijek, będziemy je pisać (podobnie jak pętle **for**) w **lewym okienku** interpretera `repl.it` (w pliku z kodem źródłowym).

Przykład Napiszmy funkcję, która wypisuje swój argument podniesiony do kwadratu:

```
def kwadrat(x):
  print(x * x)
```

```
> kwadrat(7)
49
> kwadrat(2 + 3)
25
```

Spróbuj przepisać definicję funkcji `kwadrat` do lewego okienko interpretera repl.it i nacisnąć przycisk „run”. Jeśli wszystko się powiedzie, od tej pory będziesz mógł *wywoływać* funkcję `kwadrat`, pisząc w prawym okienku `kwadrat(argument)`, podając dowolną liczbę (lub wyrażenie) jako `argument`.

Polecenia, takie jak wywołanie funkcji, możemy również zapisywać w lewym okienku, poniżej definicji tej funkcji. Wówczas, po naciśnięciu przycisku „run”, w prawym okienku zobaczymy efekt działania tych poleceń:

```
def kwadrat(x):
  print(x * x)

kwadrat(7)
for a in [2, 5, 13]:
  kwadrat(a)
```

```
49
4
25
169
```

Często chcemy aby funkcja zamiast wypisać wynik swojego działania na ekran zwróciła go w taki sposób aby można było go zapisać do jakiejś zmiennej, możliwe to jest poprzez zastosowanie instrukcji **return**. Przerywa ona działanie funkcji w miejscu w którym została wykonała, powoduje powrót do miejsca gdzie wywołana została funkcja i zwraca podaną do niej wartość:

```
def dodaj(a, b):
  return a + b

a = dodaj(7, 8)
print(a - 2)
```

```
13
```

Zadanie 3 *Napisz funkcję* ***`znak(liczba)`*** *która wypiszę informację o znaku podanej liczby (wyróżniając zero) i zwróci jej wartość bezwzględną. Wywołanie funkcji* ***`znak`*** *powinno wyglądać następująco:*

```
a = znak(7)
b = znak(-13)
c = znak(0)
print(a, b, c)
```

```
dodatnia
ujemna
zero
7 13 0
```

3.5.1 Argumenty domyślne i nazwane

Możliwe jest podanie wartości domyślnych dla wybranych argumentów funkcji, tworząc z nich argumenty opcjonalne, które nie muszą być podawane przy wywołaniu funkcji. Argumenty z wartościami domyślnymi muszą występować w definicji funkcji po argumentach bez takich wartości. Przy wywołaniu funkcji można odwoływać się do jej argumentów z podaniem ich nazw, pozwala to na podawanie argumentów w innej kolejności niż podana w definicji funkcji, co jest przydatne zwłaszcza przy funkcjach z wieloma argumentami opcjonalnymi.

```
def potega(a = 2, b = 2):
    return a ** b

print( potega(), potega(4), potega(4, 3) )
print( potega(b = 3), potega(b = 1, a = 4) )
```

```
4 16 64
8 4
```

3.5.2 Zasięg zmiennej

W Pythonie wewnątrz funkcji widoczne są zmienne zdefiniowane poza nią, jednak aby móc modyfikować taką zmienną wewnątrz funkcji należy ją tam zadeklarować jako globalną przy pomocy słowa kluczowego **global**:

```
def dodaj():
  global b
  a, b = 5, 13
  print(a, b, c)
a, b, c = 1, 3, 7
dodaj()
print(a, b, c)
```

```
5 13 7
1 13 7
```

Analizując działanie powyższego kodu zwrócić uwagę na:

- zasłonięcie globalnego `a` poprzez lokalne a wewnątrz funkcji (nie można zmodyfikować globalnej zmiennej `a` w funkcji),
- możliwość dostępu do globalnych zmiennych w funkcji dopóki ich nie zasłonimy zmienną lokalną (tak używamy zmiennej `c`)
- możliwość zmodyfikowania zmiennej globalnej gdy jest zadeklarowana w funkcji jako **global**

4 Napisy (i komentarze)

Do tej pory używaliśmy zmiennych do przechowywania liczb i operowania na nich. Zmienne mogą również jako wartości przyjmować litery, słowa, a nawet całe zdania:

```
x = 'A' # komentarz
a, b, c = 'Ala', "ma", " kota i psa"
d = """ ... a co ma ...
 "kotek"?"""
print(x, a[2])
print(c[1], c[-1], c[-3])
print(a + b)
print(3 * a)
print(a + " " + b + c + d)
```

```
A a
o a p
Alama
AlaAlaAla
Ala ma kota i psa ... a co ma ...
 "kotek"?
```

Zwróć uwagę na następujące rzeczy:

- Napisy muszą być otoczone pojedynczymi apostrofami lub podwójnym cudzysłowami (nie ma znaczenia, którą wersję wybierzemy), w przypadku napisów wieloliniowych używamy trzykrotnie apostrofu lub cudzysłowowa na początku i końcu napisu.
- Tekst występujący po znaku # stanowi **komentarz** i jest ignorowany przez interpreter, jako komentarze bywają używane także napisy wieloliniowe nie przypisywane do żadnej zmiennej.
- Przy użyciu liczby w nawiasie kwadratowym możemy poznać poszczególne litery słowa (***numeracja rozpoczyna się od 0***).
- Ujemny indeks oznacza odliczanie liter od końca słowa: ostatnia litera słowa `c` to `c[-1]`, przedostatnia to `c[-2]`, itd.
- Przy użyciu znaku dodawania możemy sklejać (*konkatenować*) napisy.
- Przy użyciu znaku gwiazdki możemy mnożyć napisy (czyli sklejać same ze sobą).

Innymi przydatnymi operacjami na słowach jest sprawdzanie długości słowa poleceniem **len()** oraz wycinanie podsłowa przy użyciu dwukropka:

```
tekst = 'Python'
dlugosc = len(tekst)
print(dlugosc, tekst[2:5], tekst[3:], tekst[:3])
```

```
6 tho hon Pyt
```

W powyższym przykładzie:

- komenda `tekst[2:5]` zwraca podnapis od znaku nr 2 (**włącznie**) do znaku nr 5 (**wyłącznie**),
- komenda `tekst[3:]` zwraca podnapis od znaku nr 3 (**włącznie**) do końca,
- komenda `tekst[:3]` zwraca podnapis od początku do znaku nr 3 (**wyłącznie**).

Podobnie jak w **range()** możemy podać trzeci argument określający przedział czyli krok. Pozwala to na wybieranie co n-tego znaku z napisu, zarówno zaczynając od początku jak i końca:

```
tekst = '123456789'
dlugosc = len(tekst)
print(tekst[::2], tekst[1::2])
print(tekst[::-1], tekst[::-3])
print(tekst[::-1][::3], tekst[::3][::-1])
```

```
13579 2468
987654321 963
963 741
```

W powyższym przykładzie:

- komenda `slowo[::2]` zwraca co drugi znak,
- komenda `slowo[1::2]` zwraca co drugi znak od znaku nr 1,
- komenda `slowo[::-1]` zwraca napis od tyłu,
- komenda `slowo[::-3]` zwraca co 3 znak z napisu od tyłu (warto zauważyć że nie zawsze jest to równoważne wypisaniu napisu złożonego z co 3 znaku od tyłu).

4.1 Napis jako lista

Wszystkie listy, których do tej pory używaliśmy w pętli **for** były listami liczb. Okazuje się, że w Pythonie napisy mogą być traktowane jako lista, a dokładniej listą liter. Oznacza to, że po słowie można przejść przy użyciu pętli **for**, tak samo jak przechodziliśmy po liście liczb:

```
for l in 'Abc':
    print('litera', end = ' ')
    print(l)
```

```
litera A
litera b
litera c
```

4.1.1 Modyfikowalność napisów

Python pozwala odwoływać się do poszczególnych znaków w napisie jak do elementów listy, jednak nie pozwala na ich modyfikowanie:

```
s = "abcdefgh"
s[2] = "X"
print(s)
```

```
Traceback (most recent call last):
  File "python", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Zwróć uwagę na komunikat błędu, który został wyświetlony, podaje on informacji o tym co wywołało błąd (opis błędu) i w której linii programu on wystąpił. **Czytanie ze zrozumieniem komunikatów o błędach ułatwia naprawianie niedziałającego programu.**

Jeżeli zachodzi potrzeba modyfikowania napisu konkretnych znaków w napisie możemy użyć poznanej wcześniej metody uzyskiwania podnapisów:

```
s = "abcdefgh"
s = s[:2] + "X" + s[3:5] + s[6:]
print(s)
```

```
abXdegh
```

Powyższy przykład w miejsce znaku nr 2 wstawia napis "X" oraz usuwa znak nr 5 z napisu. Przy konieczności modyfikacji znak po znaku możemy użyć iteracji po napisie i budować nowy napis znak po znaku:

```
s, ns = "abcdefgh", ""
for z in s:
    if z in "cf":
        ns = ns + "X"
    else:
        ns = ns + z
print(ns)
```

```
abXdeXgh
```

Zadanie 4 *Napisz funkcję* ***wyiksuj(napis)****, która wypisze dany* ***napis****, zastępując każdą małą literę polskiego alfabetu małą literą x i każdą wielką literę polskiego alfabetu wielką literą X, natomiast resztę znaków pozostawi bez zmian. Np. dla napisu* `'Python 3.6.1 (default, Dec 2015, 13:05:11)'` *program powinien wypisać:* `Xxxxxx 3.6.1 (xxxxxxx, Xxx 2015, 13:05:11)`

4.2 Konwersje liczba – napis

Z punktu widzenia komputera liczba czy też element napisu, którym jest litera są pewną wartością numeryczną. Natomiast my do zapisu liczb używamy różnych systemów (np. dziesiętnego, czy też szesnastkowego). Domyślnie liczby wprowadzane do programu interpretowane są jako zapisane w systemie dziesiętnym. Możliwe jest jednak wprowadzanie liczb zapisanych w innych systemach liczbowych lub konwersja z napisu zawierającego liczbę (drugi argument **int()** pozwala określić podstawę systemu z którego konwertujemy, zero oznacza automatyczne wykrycie w oparciu o prefix):

```
# szesnastkowo
h1, h2, h3 = 0x1F, int("0x1F", 0), int("1F", 16)
# oktalnie
o1, o2, o3 = 0o17, int("0o17", 0), int("17", 8)
# binarnie
b1, b2, b3 = 0b101, int("0b101", 0), int("101", 2)

print("",h1,o1,b1, "\n",h2,o2,b2, "\n",h3,o3,b3)
```

```
 31  15  5
 31  15  5
 31  15  5
```

Możliwe jest także konwertowanie wartości liczbowej na napis w określonym systemie liczbowym:

```
a, b = 3, 13
c = (a + b) * b
s = "(" + bin(a) + " + " + oct(b) + ") * " + hex(b) + " = " + str(c)
print( s )
```

Możliwe jest także konwertowanie pomiędzy numerem znaku Unicode a napisem go reprezentującym i w drugą stronę - służą do tego odpowiednio funkcje **chr()** i **ord()** W ramach napisów można też użyć `\uNNNN`, gdzie `NNNN` jest (czterocyfrowym) numerem znaku lub po prostu umieścić dany znak w pliku kodowanym UTF8.

```
print(chr(0x21c4) + " == \u21c4 == ⇄")
print(hex(ord("⇄")), hex(ord("\u21c4")), hex(ord(chr(0x21c4))) )
```

4.3 Kodowania znaków

Python używa Unicode dla obsługi napisów, jednak przed przekazaniem napisu do świata zewnętrznego konieczne może być zastosowanie konwersji do określonej postaci bytowej (zastosowanie odpowiedniego kodowania). Służy do tego metoda `encode()` np.:

```
a = "aąbcć ... ⇄"
inUTF7 = a.encode('utf7')
inUTF8 =  a.encode() # lub a.encode('utf8')
print("'" + a + "' w UTF7 to: " + str(inUTF7) + ", w UTF8: " + str(inUTF8))
```

Zmienne typu 'bytes' oprócz przekazania na zewnątrz (np. zapisu do pliku lub wysłania przez sieć) mogą zostać także m.in. zdekodowane do napisu z użyciem metody `decode()` lub poddane dalszej konwersji np. kodowaniu base64:

```
print("zdekodowany UTF7: " + inUTF7.decode('utf7'))

import codecs
b64 = codecs.encode(inUTF8, 'base64')
print("napis w UTF8 po zakodowaniu base64 to: " + str(b64))
```

W powyższym przykładzie należy zwrócić uwagę na instrukcję **import**, która służy do załączania bibliotek pythonowych do naszego programu. W tym wypadku załączamy fragment standardowej biblioteki Pythona o nazwie `codecs`. Base64 jest jednym z kodowań pozwalających na zapis danych binarnych w postaci ograniczonego zbioru znaków drukowalnych.

4.4 Wyrażenia regularne

W przetwarzaniu napisów bardzo często stosowane są wyrażenia regularne służące do dopasowywania napisów do wzorca który opisują, wyszukiwaniu/zastępowaniu tego wzorca. Do typowej, podstawowej składni wyrażeń regularnych zalicza się m.in. następujące operatory:

```
.      - dowolny znak
[a-z]  - znak z zakresu
[^a-z] - znak z poza zakresu (aby mieć zakres z ^ należy dać go nie na początku)
^      - początek napisu/linii
$      - koniec napisu/linii

*      - dowolna ilość powtórzeń
?      - 0 lub jedno powtórzenie
+      - jedno lub więcej powtórzeń
{n,m}  - od n do m powtórzeń

()     - pod-wyrażenie (może być używane dla operatorów powtórzeń,
         a także dla referencji wstecznych)
```

Python umożliwia korzystanie z wyrażeń regularnych za pomoca modułu `re`:

```
import re

y = "aa bb cc bb dd bb ee"

if re.match(".*[dz]", y):
        print(y, "zawiera d lub z")

# zastępowanie
print (re.sub('[bc]+', "XX", y, 2))
print (re.sub('[bc]+', "XX", y))
```

Zwróć uwagę na działanie funkcji `match`, która dopasowuje wyrażenie do początku napisu (czyli tak jakby zaczynało się od `^`).

5 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")
```

6 Zmienne i ich typy

6.1 Listy

Do tej pory listy traktowaliśmy głównie jako zbiór elementów po którym iterujemy. Zastosowanie list jest jednak znacznie szersze. Lista stanowi pewnego rodzaju kontener do przechowywania innych zmiennych, w którym elementy zorganizowane są na zasadzie określenia ich (względnej) kolejności. Lista może zawierać elementy różnych typów.

Na listach możemy wykonywać m.in. operacje modyfikowania, czy też usuwania jej elementów:

```
l = ["i", "C", 0, "M"]
l[0] = "I"
del l[2]
print(l)
```

```
['I', 'C', 'M']
```

Jednak jeżeli chcemy modyfikować elementy listy iterując po niej konieczne jest iterowanie po indeksach (a nie jak dotychczas po wartościach):

```
for i in range(len(l)):
    print(l[i])
    l[i] = "q"
print(l)
```

```
I
C
M
['q', 'q', 'q']
```

Możemy także tworzyć „podlisty” przy pomocy operatora zakresów w identyczny sposób jak to zostało opisane przy napisach, np. `ll[1::2]` zwróci listę złożoną z co drugiego elementu listy `ll` zaczynając od elementu o indeksie 1.

6.1.1 Lista jako modyfikowalny napis

Listy mogą też służyć jako narzędzie do modyfikowania napisów, w tym celu można skorzystać np. z listy złożonej z liter oryginalnego napisu:

```
s = "abcdefgh"
l = list(s)
l[2] = "X"
del(l[5])
s = "".join(l)
print(s)
```

```
abXdegh
```

6.2 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 a inny z rzeczywistymi oraz że istnieje typ związany z wartościami logicznymi (`True`/`False` - zapisywane z wielkiej litery), który pozwala na przechowywanie wyniku warunków takich jak stosowane w instrukcjach **if**, czy **while**. Zauważ także że zmienna może zmienić swój typ.

6.3 Obiektowość

Jak mogliśmy zauważyć przy sprawdzaniu typów zmiennych są one klasami. Związane z tym jest m.in. to iż posiadają one metody służące do operowania na nich. Opis danego typu wraz z dostępnymi metodami można obejrzeć przy pomocy polecenia **help()**, np. **help("list")**.

W przypadku list za pomocą metod tej klasy mamy możliwość wstawiania wartości na daną pozycję, sortowania i odwracania kolejności elementów:

```
l = ["i", "m"]
l.insert(1, "c")
print(l)
l.reverse()
print(l)
l.sort()
print(l)
```

```
['i', 'c', 'm']
['m', 'c', 'i']
['c', 'i', 'm']
```

Zwróć uwagę że sortowanie i odwracanie modyfikuje istniejącą listę a nie tworzy kopii.

Zadanie 5 *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']
```

6.4 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.

```
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
```

Zadanie 6 *Napisz funkcję* **`zlicz`** *która dla podanej listy policzy powtórzenia jej elementów. Przykład użycia:*

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

Wskazówka: możesz użyć metody **`get()`** *do pobierania wartości z słownika, jeżeli w nim jest lub wartości domyślnej w przeciwnym wypadku - szczegóły zobacz w dokumentacji*

6.4.1 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 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)]
```

6.5 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.

Zadanie 7 *Zastanów się czy konstrukcję if/elif w funkcji* ***`dzialanie()`*** *można by zastąpić słownikiem, jak to ewentualnie zrobić i jakie mogłoby mieć to zalety bądź wady?*

6.6 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).

6.6.1 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:

[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).

- 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

6.6.2 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]:

```
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.

6.7 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
  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:

[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.

```
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[3]:

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

```
INFO
80
7
```

6.8 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):
        r, a, b = a, b, a + b
        yield r

ii = iter( f(8) )
for i in f(16):
```

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

```
    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
```

7 Zadania dodatkowe

Zadanie 8 *Używając dwóch pętli* **for**, *jedna wewnątrz drugiej, napisz program, który wypisze na ekranie trójkąt z siódemek, taki jak poniżej:*

```
X
XX
XXX
XXXX
XXXXX
XXXXXX
XXXXXXX
```

Zadanie 9 *Zmodyfikuj rozwiązanie zadania 8 tak aby zamiast co najmniej jednej pętli* **for** *użyć pętli* **while**.

Zadanie 10 *Zmodyfikuj rozwiązanie zadania 8 tak aby korzystało tylko z jednej pętli.*

Zadanie 11 *Napisz funkcję* `bezwzgledne(lista)`, *która dla danej listy liczb wypiszę listę wartości bezwzględnych tych liczb, tj. liczby ujemne zamieni na przeciwne, a liczby nieujemne pozostawi bez zmian. Poszczególne liczby powinny być oddzielone pojedynczymi spacjami. Przykładowe użycie funkcji powinno wyglądać następująco:*

```
 > bezwzgledne([5, -10, 15, 0])
5 10 15 0
```

Zadanie 12 *Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy wspak.*
Np. dla listy `['Ala', 'ma', 'kota']` *funkcja powinna wypisać:*

```
alA
am
atok
```

Zadanie 13 *Napisz funkcję* `toStr(liczba, podstawa)`, *która konwertuje podaną liczbę do reprezentacji napisowej w systemie o podanej podstawie.*
Wskazówka: do testowania poprawności działania możesz użyć funkcji **int** `(napis, podstawa)`, *możemy przyjąć że podstawa jest mniejsza od 37 tak aby starczyło liter alfabetu łacińskiego.*

Zadanie 14 *Napisz funkcję* `sortuj(lista)` *która zwróci posortowaną listę. Funkcja nie może zmodyfikować oryginalnej listy.*

Zadanie 15 *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 16 *Zmodyfikuj rozwiązanie zadania 8 tak aby zamiast co najmniej jednej z pętli użyć rekurencji. Wskazówka: Funkcja rekurencyjna to funkcja, która wywołuje samą siebie (typowo ze zmodyfikowanymi argumentami), dopóki zachodzi jakiś ustalony warunek (typowo zależny od argumentów).*

8 Praca domowa

Rozwiązania zadań domowych należy przesłać na adres licealisci.pracownia@icm.edu.pl wpisując jako temat wiadomości `gx PDy`, gdzie `x` to numer grupy a `y` to numer kolejny pracy domowej, np. `g3 PD1` dla pierwszej pracy domowej w grupie 3, itd. Zadania domowe są nie obowiązkowe, jednak zachęcamy do ich robienia i wysyłania rozwiązań (nawet niekompletnych). 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 `gx`, gdzie `x` to numer grupy.

Prosimy o załączanie rozwiązań (kodu programów) w postaci załączników lub odnośnika do rozwiązań na repl.it, w tym drugim przypadku prosimy dodatkowo o wklejenie kodu w treść wiadomości.

Termin nadsyłania zdań domowych ..... (PD1) to ..... godz. 16^{59}, a zadań ..... (PD2) to ..... godz. 16^{59}.

Zadanie domowe 1 (1 pkt) *Napisz pętlę, która wypisze wszystkie dwucyfrowe liczby podzielne przez 7. Kolejne liczby powinny być wypisane w jednym wierszu i porozdzielane pojedynczymi spacjami.*

Zadanie domowe 2 (2 pkt) *Napisz funkcję, która dla danej listy słów wypisze w kolejnych wierszach ich skróty w postaci* `<pierwsza litera>-<ostatnia litera> (<dlugosc slowa>)`.
Np. dla listy `['Interdyscyplinarne', 'Centrum', 'Modelowania']` *powinna wypisać:*

```
I-e (18)
C-m (7)
M-a (11)
```

Wskazówka: wynik funkcji **len()** *mierzącej długość słowa jest liczbą. Do rozwiązania tego zadania może Ci się przydać konwersja tej liczby na słowo (aby dało się ją skleić z innymi słowami), z użyciem funkcji* **str()**

Zadanie domowe 3 (3 pkt) *Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy powtarzając każdą małą literę dwukrotnie. Np. dla* `['Ala', 'ma', 'kota', 'i PSA']` *funkcja powinna wypisać:*

```
Allaa
mmaa
kkoottaa
ii PSA
```

Zadanie domowe 4 (1 pkt) *Korzystając z metod klasy* **list** *i/lub funkcji* **sorted()** *napisz funkcję która sortuje podaną listę w kolejności malejącej.*

Zadanie domowe 5 (2 pkt) *Napisz program dekodujący napis kodowany w UTF8 zakodowany przy pomocy base64 mający postać:* `b'UHl0aG9uIGplc3QgZmFqbnkg8J+Yjg==\n'`.
Wskazówka: dane wejściowe funkcji `decode()` *muszą być typu "bytes", można to uzyskać poprzedzając napis prefiksem* `b`*, tak jak powyżej.*

Zadanie domowe 6 (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'}
```

9 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).