Linux i Python w Elektronicznej Sieci #02: Wprowadzenie do programowania w Pythonie

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

2021-03-11

1 Wprowadzenie

Python jest wysokopoziomowym językiem programowania ogólnego przeznaczenia. Oznacza to że jego składnia została tak zbudowana aby maksymalizować czytelność kodu dla człowieka i być niezależna od sprzętowych i implementacyjnych detali oraz że nie ma pojedynczego dedykowanego obszaru zastosowań (z łatwością może być stosowany do różnych zastosowań).

Wsórd cech i zalet Pythona należy wymienić:

- jest językiem interpretowanym[1], co daje łatwiejsze modyfikowanie kodu, eksperymentowanie z nim, itd
- jest jednym z najpopularniejszych języków programowania (wg niektórych źródeł nawet najpopularniejszym), więc nie jest ”dydaktyczną egzotyką”, która potem do niczego się nie przyda
- działa na wielu różnych platformach sprzętowych i na różnych systemach oparacyjnych
- istnieje bardzo wiele bibliotek pythonowych (posiadających pythonowe API), a można korzystać także z ”niedostosowanych” do Pythona bibliotek C (.so, .dll)
- jest łatwo rozszerzalny przy pomocy (własnych) bibliotek/modułów tworzonych w C/C++ (podstawowy interpreter napisany jest C)
- kod pythonowy może być łatwo wywoływany z poziomu C/C++, co pozwala na łatwe wykorzystanie Pythona jako języka skryptowego dla projektów tworzonych w C/C++

Ponadto Python jest wygodniejszy w uczeniu od wielu innych języków m.in. ze względu na to że kod pythonowy realizujący tą samą funkcjonalność, przy takim samym poziomie obsługi błędów, etc i podobnej czytelności, praktycznie zawsze jest krótszy od kodu C (a potrafi być krótszy kilkukrotnie, więc łatwiej go pokazać i omówić).

W ramach kursu zajmować się będziemy językiem Python w wersji 3 (czyli o pełnym numerze zaczynającym się od 3, np. 3.7.1). Należy o tym pamiętać i zwracać na to uwagę, gdyż wersja ta różni się na tyle znacząco w stosunku co do starszej, lecz wciąż używanej wersji 2, że programy, prezentowane w tym skrypcie i te które będziemy pisać na zajęciach nie będą działać w drugiej wersji Pythona.

Pomimo, iż Pythona można używać na różnych systemach operacyjnych (a nawet on-line), to w ramach tego kursu, jako że poznajemy na nim środowiska „unixowate”, będziemy używać go w środowisku GNU/Linux[2].

1.1 Praca z konsolą interaktywną

Pierwszym sposobem pracy z Pythonem jest praca w interaktywnej konsoli. Uzyskujemy ją po uruchomieniu polecenia `python3`. W konsoli tej początkowo wypisane są pewne informacje (m.in. używana wersja Pythona) oraz znak zachęty (w Pythonie najczęściej >>>)[3]. Interpreter oczekuje, iż po tym znaku wpiszemy polecenie i naciśniemy Enter. Wynik polecenia zostanie wypisany w kolejnym wierszu.

1. może być i w niektórych sytuacjach podlega kompilacji do kodu pośredniego celem zwiększenia wydajności
2. dokładniej najpopularniejszy jego interpreter – napisany w języku C – *CPython*
3. Zauważ że jest on inny niż znak zachęty bash'a (zazwyczaj $ poprzedzony dodatkowymi informacjami) – pozwala to na identyfikację interpretera poleceń w którym aktualnie pracujemy i wydawanie w odpowiedniej składni (bash nie rozumie poleceń w składni pythona, python nie rozumie poleceń w składni basha).

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 dzielenie.
- Znak // oznacza dzielenie całkowite.
- Znak % oznacza branie reszty z dzielenia.
- Nawiasy okrągłe służą grupowaniu wyrażeń i wymuszaniu innej niż standardowa kolejności działań.
- Spacje nie mają znaczenia (używamy ich jedynie dla zwiększenia czytelności).

Porada

W konsoli interaktywnej przy pomocy strzałek góra/dół można przeglądać historię wydanych poleceń. Polecenia te można także wykonać ponownie (naciskając enter), a przedtem także zmodyfikować (poruszając się strzałkami prawo lewo).
Konsola ta posiada także mechanizm dopełniania wpisywanych poleceń przy pomocy tabulatora (pojedyncze naciśnięcie dopełnia, gdy tylko jedna propozycja, podwójne wyświetla propozycje dopełnień).

1.1.1 Zmienne

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 $(x^2 + y^2)$.

1.1.2 Moduły i zaawansowany kalkulator

Python pozwala na wykonywanie bardziej zaawansowanych obliczeń. Możliwe jest m.in. obliczenia wartości wyrażeń logicznych, konwertowanie systemów liczbowych, obliczanie wartości funkcji trygonometrycznych.

Duża część funkcji matematycznych w Pythonie zawarta jest w module „math”, który wymaga zaimportowania. Można to zrobić na przykład w sposób następujący:

```
>>> import math
>>> math.sin(math.pi/2)
1.0
```

Zauważ, że odwołanie do elementów tak zaimportowanego modułu wymaga podania jego nazwy, następnie kropki i nazwy używanej funkcji z tego modułu.

1.2 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 i uruchamianie tego kodu w konsoli.

Moduły

Nazwa pliku powinna być inna niż nazwy importowanych modułów, czyli jeżeli w kodzie mamy `import abc` to nasz plik nie powinien nazywać się `abc.py`, w przeciwnym razie zamiast wskazanego modułu Python będzie próbował zaimportować nasz plik.

Utwórz plik `mojProgram.py`[4] z następującą zawartością:

```
x = 3
y = 4
print(x**2 + y**2)
```

W celu wykonania kodu zapisanego w pliku uruchom interpreter Pythona z jednym argumentem, będącym nazwą tego pliku: `python3 mojProgram.py`.

Porada

Zachowuj pliki z programami pisanymi w trakcie zajęć, używając nazw które pozwolą Ci łatwo zidentyfikować dany program. Mogą one być pomocne w rozwiązywaniu kolejnych zadań oraz prac domowych.

1.2.1 funkcja `print`

Zwróć uwagę, iż do wypisania wyniku działania na ekran została użyta funkcja **`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 właśnie np. za pomocą funkcji **`print`**.

Funkcja **`print`** wypisuje przekazane do niej (rozdzielane przecinkami) argumenty rozdzielając je spacjami. Przechodzi ona domyślnie do następnej linii po każdym wywołaniu. Na przykład:

```
print("raz dwa", "trzy ...")
print(4, 5)
```

```
raz dwa trzy ...
4 5
```

4. Pliki z skryptami Pythona tradycyjnie mają rozszerzenie `.py`. Nie jest ono jednak wymagane — interpreter Pythona wykona kod z pliku o dowolnym rozszerzeniu a także z pliku bez rozszerzenia.

Informacja

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:

Zachowanie funkcji `print` można zmienić, dodając do jej wywołania, na końcu listy argumentów argument postaci `end = X` i/lub `sep = Y`, gdzie `X` to otoczony apostrofami ciąg znaków, który chcemy wypisywać zamiast przejścia do nowej linii, a `Y` to otoczony apostrofami ciąg znaków, który chcemy wypisywać zamiast spacji rozdzielającej wypisania kolejnych argumentów. Na przykład:

```
x = 3
y = 4
print(x, '+ ', end='')
print(y, x + y, sep=' = ')
```

```
3 + 4 = 7
```

Napisy

Ciąg znaków ujęty w apostrofy lub cudzysłowy (w Pythonie nie ma znaczenia, której wersji użyjemy, ważne jest tylko aby znak rozpoczynający i kończący był taki sam) nazywamy napisem. Możemy ich używać nie tylko w ramach funkcji print, ale też np. przypisywać do zmiennych. Więcej o napisach dowiemy się później.

1.2.2 Komentarze

Często chcemy móc umieścić w kodzie programu dodatkową informację, która ułatwi nam jego czytanie i zrozumienie w przyszłości. Służą do tego tak zwane komentarze, które są ignorowane przez interpreter (bądź kompilator) danego języka. W Pythonie podstawowym typem komentarza, jest komentarz jednoliniowy, rozpoczynający się od znaku # a kończący z końcem linii.

1.2.3 inne sposoby uruchamiania kodu z pliku

Jeżeli do wywołania interpretera Pythona dodamy opcję `-i` (np. `python3 -i mojProgram.py`) po wykonaniu kodu z podanego pliku uruchomi on konsolę interaktywną w której będą dostępne elementy (m.in. zmienne) zdefiniowane w podanym pliku.

Możliwe jest także włączenie kodu z pliku do aktualnie uruchomionego interpretera (np. konsoli interaktywnej), w taki sposób jakbyśmy go wpisali (czyli z wykonaniem wszystkich instrukcji i późniejszą możliwością dostępu do zdefiniowanych tam elementów). Aby wczytać w ten sposób kod z pliku `mojProgram.py` należy wykonać: `exec(open('mojProgram.py').read())`

1.2.4 ipython

`ipython3` jest wygodniejszym w pracy interaktywnej interpreterem Pythona w wersji 3. Pozwala on m.in. na lepsze przewijanie i edytowanie poleceń wieloliniowych w historii.

2 Podstawowe elementy składniowe

2.1 Definiowanie własnych funkcji

Bardzo często będziemy chcieli móc 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 kilka rzeczy:

- Na końcu pierwszej linijki jest dwukropek.
- Druga linijka musi być *wcięta*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji.
- Jeżeli w ramach funkcji chcemy wykonać kilka instrukcji muszą one mieć taki sam poziom wcięcia.
- „Wnętrze" funkcji kończymy wracając do takiego samego poziomu wcięcia na jakim ją rozpoczęliśmy (takiego wcięcia jakie miała linijka z słowem kluczowym `def`).

Jest to typowy sposób wyznaczania bloku kodu w Pythonie i będziemy go jeszcze spotykać w innych konstrukcjach (które poznamy już niedługo), dlatego szczególnie wart jest zapamiętania.

Gdy umieszczamy inną konstrukcję korzystającą z bloku kodu we wnętrzu jakiegoś innego bloku (np. funkcji), blok tej instrukcji musi być „bardziej" wcięty od bloku w którym jest zawarty, powrót do poziomu wcięcia zewnętrznego bloku oznacza zakończenie bloku tej instrukcji i kontynuowanie zewnętrznego bloku.

Porada

Na funkcję można patrzeć jak na nazwany kawałek kodu, który możemy wywołać z innego miejsca ze odmiennymi wartościami zmiennych stanowiących jej argumenty.

Polecenie wywołania funkcji ma postać `nazwa_funkcji(argumenty)` i możemy napisać je w tym samym pliku, poniżej definicji tej funkcji. Typowo ilość i kolejność argumentów w definicji, jak i w wywołaniu powinny być takie same. Jeżeli nasza funkcja nie potrzebuje przyjmować argumentów nawiasy okrągłe w jej definicji i wywołaniu pozostawiamy puste. Jeżeli potrzebujemy więcej argumentów rozdzielamy je w obu przypadkach przecinkami (tak jak miało to miejsce w korzystaniu z funkcji `print`).

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

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

kwadrat(7)
kwadrat(2 + 3)
```

```
49
25
```

Zwróć uwagę, iż wywołania funkcji w powyższym przykładzie nie są wcięte — są poza blokiem funkcji.

Polecenia wieloliniowe w konsoli interaktywnej

Możliwe jest wprowadzanie poleceń wieloliniowych w konsoli interaktywnej. W takim wypadku po wprowadzeniu pierwszej linii (rozpoczynającej blok, np. `def`) nastąpi zmiana znaku zachęty na ..., co oznacza tryb wprowadzania bloku poleceń. Następnie wprowadzamy kolejne instrukcje wykonywane w ramach tego bloku (np. funkcji) pamiętając o wcięciach. Wprowadzanie bloku kończymy pustą linią, po czym znak zachęty powróci do standardowego >>>.

Dla znających C lub C++ (1/2)

- kolejne instrukcje (zamiast średnika) kończy znak nowej linii
- średnik na końcu instrukcji (linii) nie jest błędem składniowym (jest ignorowany)
- bloki rozpoczyna dwukropek, a wyznacza je wcięcie o danej ilości znaków (nie mieszamy tabulatorów z spacjami)

2.1.1 Wartość zwracana z funkcji

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 kwadrat(x):
  return x * x

a = kwadrat(7)
print( a - 2, kwadrat(4) )
```

```
47 16
```

2.1.2 Argumenty domyślne i nazwane

Możliwe jest podanie wartości domyślnych dla wybranych argumentów funkcji. Utworzy to 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
```

2.1.3 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 test():
  global b
  a, b = 5, 13
  print(a, b, c)

a, b, c = 1, 3, 7
test()
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`

2.2 Pętla `for`

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

```
print(1 * 1)
print(2 * 2)
print(3 * 3)
print(4 * 4)
```

```
1
4
9
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 i uruchomić program. Zauważ że wnętrze pętli jest wyznaczone w sposób analogiczny do wnętrza funkcji:

- Rozpoczyna się od dwukropka kończącego pierwszą linię.
- Kolejne linijki są *wcięte*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji.
- Jeżeli w ramach pętli chcielibyśmy wykonać kilka instrukcji muszą one mieć taki sam poziom wcięcia.
- „Wnętrze” pętli kończymy wracając do takiego samego poziomu wcięcia na jakim ją rozpoczęliśmy (takiego wcięcia jakie miała linijka z słowem kluczowym **for**).
- Pętle możemy zagnieżdżać jedna w drugiej — blok wewnętrznej pętli musi być „bardziej” wcięty. Powrót do poziomu wcięcia zewnętrznej pętli oznacza zakończenie pętli wewnętrznej i kontynuowanie zewnętrznej.

2.3 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.

2.4 Typ logiczny

Jak już się przekonaliśmy można używać Pythona jako kalkulatora. Możemy go także użyć do obliczania wartości wyrażeń logicznych. Służy do tego wbudowany dwuwartościowy typ logiczny z wartościami:

- `True` oznaczającą logiczną jedynkę / prawdę
- `False` oznaczającą logiczne zero / fałsz

Operacje na tym typie wykonujemy z użyciem słów kluczowych: **`and`**, **`or`**, **`not`** oznaczających odpowiednio: iloczyn logiczny (aby był prawdą oba warunki muszą być spełnione), sumę logiczną (aby wynik był prawdą co najmniej jednej z warunków musi być spełniony) oraz negację logiczną. Podobnie jak w zwykłych operacjach arytmetycznych możemy grupować ich fragmenty (celem wymuszenia kolejności działań) przy pomocy nawiasów okrągłych.

Wartościom tego typu mogą odpowiadać wybrane wartości innych typów (np. liczba całkowita 0 odpowiada `False`, a pozostałe liczby całkowite `True`). Wartościami tego typu są też wyniki różnego rodzaju porównań, takich jak: `<` (mniejsze), `>` (większe), `<=` (mniejsze równe), `>=` (większe równe), `==` (równe), `!=` (nierówne).

2.5 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 funkcję, która dla podanej wartości sprawdzi czy odpowiada ona logicznej prawdzie czy fałszowi i wypisuje odpowiedni komunikat. Zatem kod będzie wyglądał następująco:

```
def sprawdz(x):
    if x:
        print(x, '-- prawda')
    else:
        print(x, '-- nie prawda')
sprawdz(1)
sprawdz(0)
```

```
1 -- prawda
0 -- nie prawda
```

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 **`def`**) kończą się dwukropkiem.
- „Wnętrze” **`if`**-a i **`else`**-a (linijki 3 i 5) jest wcięte (bardziej niż samo wnętrze definicji funkcji `sprawdz`).
- Linijka 3 zostanie wykonana, jeśli spełniony będzie warunek z linijki 2, czyli jeśli wartość zmiennej x będzie odpowiadała prawdzie.
- Linijka 5 zostanie wykonana, jeśli warunek z linijki 2 nie będzie spełniony.

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, że w roli warunków sprawdzanych w ramach **if**a mogą występować bardziej złożone wyrażenia. Możemy tutaj użyć dowolnego wyrażenia którego wynik odpowiada wartości logicznej `True`/`False`, najczęściej spotkamy się z wyrażeniami złożonymi z poznanych już operatorów porównań (`<`, `>`, `<=`, `>=`, `==`, `!=`) i operacji logicznych (**and**, **or**, **not**).

Zwróć uwagę na warunek postaci „`A in B`”. Taki warunek sprawdza, czy wartość reprezentowana przez `A` jest elementem `B`, a jego wynik oczywiście także jest wartością logiczną. 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**/**elif** wykonany zostanie jedynie kod związany z pierwszym pasującym warunkiem.

Dla znających C lub C++ (2/2)

- nie ma konstrukcji i++, czy też ++i, jest za to i += 1
- warunek if'a w nawiasach nie jest błędem składniowym (ale po nawiasach musi być dwukropek)
- nie ma pętli for w stylu C („trójinstrukcyjnej”), w Pythonie pętla for zawsze iteruje po elementach jakiejś listy

2.6 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.

Pętla **while** jest też naturalnym wyborem gdy w Pythonie chcemy przechodzić przez jakiś zakres liczb z krokiem nie całkowitym (wcześniej poznana instrukcja `range`, stosowana do iterowania po zakresie liczbowym w pętli **for**, wymaga aby krok był całkowity).

Zauważ że pętla będzie się wykonywała dopóki warunek jest spełniony, zatem łatwo przy jej pomocy stworzyć pętlę nieskończoną – zarówno celowo jak i w wyniku błędu. Dlatego należy mieć na uwadze ryzyko zapętlenia (nieskończonego wykonywania takiej pętli na skutek jakiejś pomyłki).

2.7 break i continue

W ramach zarówno pętli for jak i while możemy użyć instrukcji:

- **break** powodującej przerwanie wykonywania pętli
- **continue** powodującej pominięcie pozostałych instrukcji w aktualnym obiegu pętli

Ich działanie może zobrazować poniższy kod:

```
for x in [1, 2, 3, 4, 5, 6]:
  print("start", x)
  if x == 2:
    continue
  if x == 4:
    break
  print("...")
```

```
start 1
...
start 2
start 3
...
start 4
```

2.8 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 czym 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 Napisy

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'
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.
  Nie przypisane do żadnej zmiennej napisy wieloliniowe mogą być stosowane jako komentarze wieloliniowe.
- Przy użyciu liczby w nawiasie kwadratowym możemy poznać poszczególne litery napisu (***numeracja rozpoczyna się od 0***).
- Ujemny indeks oznacza odliczanie liter od końca napisu: ostatnia litera napisu `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 napisach jest sprawdzanie długości napisu poleceniem `len()` oraz wycinanie podnapisu 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'
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 `tekst[::2]` zwraca co drugi znak,
- komenda `tekst[1::2]` zwraca co drugi znak od znaku nr 1,
- komenda `tekst[::-1]` zwraca napis od tyłu,
- komenda `tekst[::-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).

3.1 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 napisie 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
```

3.1.1 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
```

3.2 Obiektowość

Jak być może zauważyliśmy wszystkie podstawowe typy w Pythonie są klasami. Związane z tym jest m.in. to iż posiadają one metody służące do operowania na nich. Metodą nazywamy funkcję związaną z danym typem i wykonywaną na obiekcie tego typu. Zapisywane jest to z użyciem kropki, nazwy metody i nawiasów okrągłych które mogą zawierać dodatkowe argumenty. Na przykład: `"aącd".islower()` jest wywołaniem metody `islower` typu napisowego na napisie `"aącd"`; metoda ta sprawdza czy w podanym ciągu znaków nie występują wielkie litery.

Klasy posiadają także konstruktory, które możemy wywołać używając nazwy danej klasy jak funkcji i użyć np. do konwersji pomiędzy różnymi typami. Jak już wiemy wszystkie nazwy w pythonie żyją w jednym świecie, dotyczy to też nazw klas. Dlatego warto uważać aby nie nazywać swoich zmiennych zarówno tak jak nazywają się wbudowane funkcje, ani tak jak nazywają się wbudowane typy danych (takie jak int, bool, str, folat i tak dalej).

Opis danego typu wraz z dostępnymi metodami można obejrzeć przy pomocy polecenia `help()`, np. `help("str")`.

W przypadku napisów za pomocą metod tej klasy mamy możliwość między innymi wyszukania miejsca wystąpienia podnapisu, zamiany wielkich liter na małe i odwrotnie, etc.

3.3 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, podobnie liczby uzyskiwane poprzez konwersję napisu przy pomocy funkcji `int()` (dokładniej jest to konstruktor typu całkowitego). Możliwe jest jednak wprowadzanie liczb zapisanych w innych systemach liczbowych lub konwersja z napisu zawierającego liczbę — drugi, opcjonalny 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 )
```

3.4 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 = "ąąbcć ... ←↓→"
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, co pozwala m.in. na osadzanie danych binarnych (np. obrazki) w plikach tekstowych (np. dokumenty html, pliki źródłowe programów).

3.4.1 Konwersja pomiędzy znakiem a jego numerem

Możliwe jest także konwertowanie pomiędzy liczbowym 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` lub `\UNNNNNNNN` (gdzie `NNNN`/`NNNNNNNN` jest cztero/ośmio[5] cyfrowym numerem znaku zapisanym szesnastkowo) lub po prostu umieścić dany znak w pliku kodowanym UTF8[6].

5. Użycie wariantu cztero cyfrowego jest możliwe jedynie dla znaków unicode o numerach mniejszych niż 0xffff
6. Użyty w przykładzie symbol nieskończoności można uzyskać na standardowej polskiej klawiaturze pod Linuxem przy pomocy kombinacji AltGr + Shift + M

```
print(chr(0x221e) + " == \u221e == ∞ == \U0000221e")
print(hex(ord("∞")), hex(ord("\u221e")), hex(ord(chr(0x221e))) )
```

Zwróć uwagę na równoważność poszczególnych zapisów – wypisują taki sam znak na konsoli, zwracają taki sam numer unicodowy.

Niektóre znaki specjalne jak np. znak nowej linii, tabulator możemy wprowadzić z użyciem krótszych i łatwiejszych do zapamietania sekwencji niż opartych o ich numer. Dla znaku nowej linii jest to \n, a tabulatora \t.

3.5 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)
|       - alternatywa: wystąpienie wyrażenia podanego po lewej stronie
          albo wyrażenia podanego prawej stronie
```

Python umożliwia korzystanie z wyrażeń regularnych za pomocą modułu re:

```
import re
y = "aa bb cc bb ff bb ee"
x = "aa bb cc dd ff gg ee"

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

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

if re.search(" ([a-z]{2}) .* \\1", y):
  print("y zawiera dwa razy to samo")

if re.search(" ([a-z]{2}) .* \\1", x):
  print("x zawiera dwa razy to samo")
```

```
x zawiera d lub z
y zawiera dwa razy to samo
```

Funkcja search zwraca więcej informacji niż sam fakt pasowania lub nie pasowania:

```
import re
x = "aa bb cc dd ff gg ee"

# wypisanie dopasowania
wynik = re.search("cc (xx)|(dd) ff", x)
if wynik:
  print( "dopasowano tekst:", wynik.group(0) )
  print( "na pozycji:", wynik.span()[0] )

wynik = re.search("cc (xx|dd) ff", x)
if wynik:
  print( "dopasowano tekst:", wynik.group(0) )
  print( "na pozycji:", wynik.span()[0] )
```

```
dopasowano tekst: dd ff
na pozycji: 9
dopasowano tekst: cc dd ff
na pozycji: 6
```

Wyrażeń regularnych możemy używać także do operacji wyszukaj i zastąp pasujący fragment napisu:

```
import re
y = "aa bb cc bb ff bb ee"

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

# zachłanność
print (re.sub('bb (.*) bb', "X \\1 X", y))
print (re.sub('.*bb (.*) bb.*', "\\1", y))
print (re.sub('.*?bb (.*) bb.*', "\\1", y))
```

```
aa XX XX bb ff bb ee
aa XX XX XX ff XX ee
aa X cc bb ff X ee
ff
cc bb ff
```

Zwróć uwagę na:

- Działanie funkcji `search`, która wyszukuje podnapis pasujący do wyrażenia i umożliwia zaróno uzyskanie pasującego podnapisu, jak też samej informacji o fakcie pasowania lub nie do wyrażenia.
- Działanie alternatywy i nawiasów - standardowo alternatywa obejmuje wszystko co po lewej kontra wszystko co po prawej, nawiasy obejmujące fragment prawej bądź lewej strony na to nie wpływają (`cc (xx)|(dd) ff` nie zadziało jako ”xx” ablo ”dd” pomiędzy ”cc” a ”ff”, a jako ”cc xx” albo ”dd ff”), aby ograniczyć działanie alternatywy tylko do fragmentu wyrażenia należy objąć nawiasami ten fragment wraz z alternatywą w nim umieszczoną (`cc (xx|dd) ff` zadziało jako ”xx” ablo ”dd” pomiędzy ”cc” a ”ff”.
- Odwołania wsteczne do pod-wyrażeń (fragmentów ujętych w nawiasy) postaci `\\x`, gdzie `x` jest numerem pod-wyrażenia.
- „*Zachłanność*” (ang. *greedy*) wyrażeń regularnych:
  - w pierwszym wypadku `bb (.*) bb` dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`,
  - w drugim przypadku gdy zostało poprzedzone `.*` dopasowało tylko `ff`, gdyż `.*` dopasowało najdłuższy możliwy fragment czyli `aa bb cc`,
  - w trzecim wypadku `bb (.*) bb` mogło i dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`, gdyż było poprzedzone niezachłanną odmianą dopasowania dowolnego napisu, czyli: `.*?`.

  Po każdym z operatorów powtórzeń (`. ? + {n,m}`) możemy dodać pytajnik (`.? ?? +? {n,m}?`) aby wskazać że ma on dopasowywać najmniejszy możliwy fragment, czyli ma działać nie zachłannie.

4 Zmienne i ich typy

4.1 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']
```

W powyższym przykładzie widzimy:

- Modyfikację pierwszego elementu listy (`l[0] = "I"`), z użyciem odwołania poprzez numer elementu. Elementy list numerujemy od zera. Ujemne wartości oznaczają numerowanie od końca listy, czyli -1 jest ostatnim elementem listy, -2 przedostatnim, itd.
- Usunięcie trzeciego elementu listy (`del l[2]`). Powoduje to zmianę numeracji kolejnych elementów.

Jednak jeżeli chcemy modyfikować elementy listy iterując po niej, to 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']
```

Dzieje się tak gdyż przypisanie do zmiennej `x` jakiejś wartości w ramach konstrukcji `for x in lista:` modyfikuje tylko zmienną `x`, a nie element listy który został do niej pobrany.

4.1.1 Wybór podlisty

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.

4.1.2 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 = str.join("", l)
print(s)
```

```
abXdegh
```

4.1.3 Obiektowość 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.

5 Wykład wideo

- *Python: wprowadzenie* – https://www.youtube.com/watch?v=Nhb4WX_9Tgc
- *Python: funkcje* – https://www.youtube.com/watch?v=uHBzNSWTGMA
- *Python: pętle i warunki* – https://www.youtube.com/watch?v=d1igYsEZymM
- *Python: napisy* – https://www.youtube.com/watch?v=A0ZtouykAjI
- *Python: wyrażenia regularne* – https://www.youtube.com/watch?v=OklkE9T8ur8
- *Python: listy* – https://www.youtube.com/watch?v=vpMGkToxf_I

6 Zadania

Informacja

Często w zadaniach programistycznych i zawsze w ramach tego kursu jeżeli jest powiedziane:

- „napisz funkcję” to znaczy że ma zostać napisana funkcja, a nie jedynie kod programu, który mógłby stanowić wnętrze (ciało) tej funkcji,
- „napisz program” to znaczy że ma zostać napisany pełny kod programu realizujący podane czynności,
- „napisz pętlę/warunek/...” to znaczy że wystarczy napisać sam kod pętli, warunku, innej konstrukcji (ale nie tylko jego wnętrze, lecz kod całej żądanej konstrukcji składniowej),
- „napisz funkcję przyjmującą napis” to znaczy że funkcja ma mieć argument, który będzie traktowany jako napis (nie oznacza to że wymaga się wczytania tego napisu „z klawiatury”[a]),
- „napisz funkcję zwracającą X” to znaczy że funkcja ma zwrócić (poprzez return) wartość określoną przez X (nie ma jej wypisywać na ekran),
- „napisz funkcję wypisującą X” to znaczy że funkcja ma wypisać na ekran (standardowe wyjście) wartość określoną przez X,

a. Powszechnie używane w nauce programowania wczytywanie danych „z klawiatury” / odpytywanie użytkownika o kolejne parametry na ogół nie jest najlepszym rozwiązaniem programistycznym, o tym dlaczego dowiesz się w dalszych częściach tego kursu

6.1 konstrukcje składniowe

Zadanie 6.1.1

Napisz funkcję, która przyjmuje dwa argumenty i zwraca ich sumę. Użyj jej do obliczenia (oraz wypisania na konsolę) wartości kilku różnych sum.
Wskazówka: `print()` powinien być użyty na zewnątrz tej funkcji.

Zadanie 6.1.2

Napisz program obliczający sumę $1^2 + 2^2 + 3^2 + ... + 99^2 + 100^2$.

Zadanie 6.1.3

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)
```

```
7 jest dodatnia
-13 jest ujemna
0 to zero
7 13 0
```

Zadanie 6.1.4

Rozwiąż zadanie 6.1.2 stosując pętlę `while`.

6.2 napisy

Zadanie 6.2.1

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
```

Wskazówka: Po elementach listy znajdującej się w zmiennej możemy iterować pętlą `for`, tak jak robiliśmy to po literach napisu, czy po elementach listy liczb zapisanej bezpośrednio w konstrukcji pętli (spróbuj `for x in lista:`).

Zadanie 6.2.2

Napisz funkcję `wyiksuj(napis)`, która zwróci dany `napis`, zastępując każdą małą literę przez `x` i każdą wielką literę przez `X`, natomiast resztę znaków pozostawi bez zmian. Np. dla napisu `'Python 3.6.1 (default, Dec 2015, 13:05:11)'` funkcja powinna zwrócić napis: `Xxxxxx 3.6.1 (xxxxxxx, Xxx 2015, 13:05:11)`

Zadanie 6.2.3

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.

6.3 wyrażenia regularne

Zadanie 6.3.1

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest słowem (tzn. nie zawiera spacji).

Zadanie 6.3.2

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest liczbą (tzn. jest złożony z cyfr i kropki, a na początku może wystąpić + albo -).

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 6.1.1

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

a = suma(17, 15)
print(a, suma(13, 16), suma(a, 11))
```

Zwróć uwagę na użycie słowa kluczowego `return` do zwrócenia wartości z funkcji. W odróżnieniu od bezpośredniego wpisania wyniku na ekran z użyciem np. `print` pozwala to m.in. na przechowanie tego wyniku w zmiennej i użycie w dalszych obliczeniach, co zostało zademonstrowane w drugiej części przykładu.

Rozwiązanie zadania 6.1.2

```
sum = 0
for x in range(101):
    sum = sum + x**2
print(sum)
```

Zwróć uwagę na użycie zewnętrznej w stosunku co do pętli zmiennej `sum`, służącej do przechowywania wartości modyfikowanej w każdym obiegu pętli (wyniku sumy) – jest to typowy schemat rozwiązywania tego typu problemów programistycznych.

Rozwiązanie zadania 6.1.3

```
def znak(liczba):
    if liczba > 0:
        print(liczba, "jest dodatnia")
        return liczba
    elif liczba < 0:
        print(liczba, "jest ujemna")
        return -liczba
    else:
```

```
    print(liczba, "to zero")
    return 0
```

Zadanie można by rozwiązać także używając funkcji obliczającej wartość bezwzględną (`abs()`), jednak ze względu na konstrukcję zadania musimy sami ustalić znak liczby, natomiast użycie tej (i tak już posiadanej) informacji do obliczenia wartości bezwzględnej jest wydaniejsze niż kolejne sprawdzanie znaku wewnątrz funkcji `abs()`.

Rozwiązanie zadania 6.1.4

```
sum = 0
x = 1
while x <= 100:
    sum = sum + x**2
    x = x + 1
print(sum)
```

Zauważ że w tym wariancie zadania potrzebujemy dwóch zmiennych zewnętrznych w stosunku co do pętli – jednej do przechowywania obliczanej sumy, a drugiej do przechowywania numeru kroku (wartości którą sumujemy). Ta druga zmienna w rozwiązaniu z użyciem pętli `for` jest dostarczana przez samą konstrukcję tamtej pętli. W przypadku pętli `while` to my jako twórcy kodu musimy ją zainicjalizować i zwiększać w każdym kroku. Pozwala to jednak na stosowanie bardziej zaawansowanych mechanizmów modyfikowania tej zmiennej.

Rozwiązanie zadania 6.2.1

```
def wskap(lista):
    for slowo in lista:
        # w pythonie zamiast ponizszej petli można prościej ...
        # ale warto poznać (także) takie rozwiązanie
        for i in range(len(slowo)):
            print(slowo[-1 - i], end = '')
        print()
```

Prostszym rozwiązaniem (nie wymagającym jawnego pisania pętli w pętli) jest:

```
def wskap(lista):
    for slowo in lista:
        print(slowo[::-1])
```

które korzysta z odwrócenia napisu przy pomocy pobrania wszystkich jego elementów z krokiem -1 poprzez `slowo[::-1]`

Rozwiązanie zadania 6.2.2

```
def wyiksuj(napis):
    duzy_alfabet = 'AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ'
    maly_alfabet = 'aąbcćdeęfghijklłmnńoóprsśtuwyzźż'
    for c in napis:
        if c in duzy_alfabet:
            print('X', end = '')
        elif c in maly_alfabet:
            print('x', end = '')
        else:
            print(c, end = '')
```

inne rozwiązanie:

```
def wyiksuj(napis):
    for c in napis:
        if c.isupper():
```

```
      print('X', end = '')
    elif c.islower():
      print('x', end = '')
    else:
      print(c, end = '')
```

jeszcze inne rozwiązanie (w tej formie obsługuje tylko litery ASCII, ale aktualna wersja zadania to dopuszcza):

```
def wyiksuj(napis):
  import re
  napis = re.sub("[a-z]", "x", napis)
  return re.sub("[A-Z]", "X", napis)
```

Zwróć uwagę że:

- iterowanie po elementach napisu (znakach) z użyciem pętli `for`
- zastosowanie konstrukcji a `in` b do sprawdzenia czy element a (w tym wypadku znak) należy do kolekcji b (w tym wypadku napisu, ale mogła by to być także np. lista znaków)
- zastosowanie metod `isupper()` i `islower()` w drugim wariancie rozwiązania, podobne porównanie dla znaków ASCI można łatwo wykonać w oparciu o wartość numerycznego kodu tego znaku
- zwięzłość rozwiązania z użyciem wyrażeń regularnych

Rozwiązanie zadania 6.2.3

```
import codecs
d = b'UHl0aG9uIGplc3QgZmFqbnkg8J+Yjg==\n'
d = codecs.decode(d, 'base64')
d = d.decode()
print(d)
```

Zakodowany tekst to: Python jest fajny 😎
Zwróć uwagę że:

- zdejmowanie kolejnych kodowań w kolejnych krokach procedury – w odwrotnej kolejności niż były nakładane
- funkcja `codecs.decode` wymaga jako danych wejściowych ciągu bajtowego, i taki ciąg zwraca
- metoda `decode` ciągu bajtowego zwraca napis powstały przez zdekodowanie tego ciągu z użyciem utf-8

Rozwiązanie zadania 6.3.1

```
import re
def spr(x):
  if re.match("[^ ]*$", x):
    print(x, "jest słowem")
  else:
    print(x, "NIE jest słowem")
```

Zadanie polega przede wszystkim na wymyśleniu odpowiedniego wyrażenia regularnego. Ze względu że funkcja `match` dopasowuje zawsze od początku napisu (ale nie wymaga dojścia do końca napisu) nasze wyrażenie musi konczyć się dolarem, aby wyrażenie było dopasowywane do całości sprawdzanego napisu. Zastosowane wyrażenie wymaga aby napis nie zawierał spacji - wtedy uznajemy go za słowo.

Rozwiązanie zadania 6.3.2

```
import re
def spr(x):
  if re.match("[+-]?[0-9.]+$", x):
```

```
    print(x, "jest liczbą")
else:
    print(x, "NIE jest liczbą")
```