Laboratorium programistyczne — Python: Wprowadzenie i wyrażenia logiczne

Projekt „Matematyka dla ciekawych świata”
Robert Paciorek

2019-02-28

1 Wprowadzenie

Na zajęciach będziemy programować w języku Python w wersji 3. Pythona można używać na różnych systemach operacyjnych (wliczając w to Linux, MacOS i Windows), a nawet on-line. Mimo to zachęcamy do spróbowania, w ramach tych zajęć, pracy w środowisku GNU/Linux.

Zwróćcie uwagę, aby korzystać z wersji Pythona o numerze rozpoczynającym się od 3 (np. 3.6.0), a nie starszej, lecz wciąż używanej wersji 2. Wersje te różnią się tak znacząco, że programy, które będziemy pisać na zajęciach nie będą działać w drugiej wersji Pythona.

Kurs ten (podzielony na kilka skryptów) jest wprowadzeniem do programowania w Pythonie i omawia wszystkie najważniejsze elementy tego języka. Zagadnienia bardziej zaawansowane, treści dodatkowe i ciekawostki zostały oznaczone ikonką 🤔 (Thinking Face Emoji).

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 `>>>`). 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 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).

Zadanie 1.1.1

Korzystając z Pythona jak z prostego kalkulatora, oblicz sumę dodatnich liczb całkowitych mniejszych od 7.

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(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.

Utwórz plik `mojProgram.py` 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[1]: `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
```

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
```

Zadanie 1.2.1

Zmodyfikuj powyższy program tak aby uzyskać ten sam efekt używając tylko jednego wywołania funkcji **print**

Napisy

Ciąg znaków ujęty w apostrofy lub cudzysłowowa (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] 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.

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())`

1.3 Inne interpretery Pythona

1.3.1 on-line

Dostępne są różne on-line'owe interpretery Pythona, np: http://ideone.com/, http://repl.it/. Mogą one posłużyć np. do odrabiania prac domowych bez konieczności instalowania Pythona na używanym do tego komputerze.

1.3.2 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.

1.4 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.

Zadanie 1.4.1

Napisz funkcję, która przyjmuje dwa argumenty i wypisuje ich sumę. Użyj jej do obliczenia (wypisania na konsolę) wartości kilku różnych sum.

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 `>>>`.

1.4.1 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
```

Zadanie 1.4.2

Napisz funkcję, która przyjmuje dwa argumenty i zwraca ich sumę. Użyj jej do obliczenia (wypisania na konsolę) wartości kilku różnych sum.

Zadanie 1.4.3

Napisz funkcję, która oblicza i zwraca pole koła o podanym promieniu. Użyj jej do obliczenia powierzchni koła o promieniu 13.

1.4.2 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
```

1.4.3 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 Operacje logiczne

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 conajmniej jednej z warunków musi być spełniony) oraz negację logiczną.

```
a = (True or False) and True
b = (True or False) and not a
print(a, b, not b)
```

```
True False True
```

Zauważ, że wartości `True` / `False` pisane są z wielkiej litery, a słowa kluczowe oznaczające operacje logiczne zapisywane są małymi literami.

2.1 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.
- Wartość 0 odpowiada fałszowi, a wartość 1 prawdzie.

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:

```
def sprawdz2(x):
    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')

sprawdz2(0)
sprawdz2(1)
sprawdz2(2)
```

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

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. Warunki można łączyć 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.

Możemy korzystać m.in. z następujących operatorów porównań: `<` (mniejsze), `>` (większe), `<=` (mniejsze równe), `>=` (większe równe), `==` (równe), `!=` (nierówne). Zwróć uwagę na warunek postaci „`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**/**elif** wykonany zostanie jedynie kod związany z pierwszym pasującym warunkiem.

Zadanie 2.1.1

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)                    7 jest dodatnia
b = znak(-13)                  -13 jest ujemna
c = znak(0)                    0 to zero
print(a, b, c)                 7 13 0
```

Zadanie 2.1.2

Napisz funkcję, przyjmującą dwa argumenty a i b, która sprawdzi warunek równoważności $a \Leftrightarrow b$ (sprawdzi czy a wtedy i tylko wtedy gdy b).

Zadanie 2.1.3

Napisz funkcję, przyjmującą dwa argumenty a i b, która sprawdzi warunek implikacji $a \Rightarrow b$ (sprawdzi czy z a wynika b).

2.2 Pętla while

Załóżmy, że chcemy sprawdzić znak wszystkich liczb od -2 do 2. Zgodnie z dotychczasową wiedzą, w tym celu musielibyśmy 5cio krotnie wywołać naszą funkcję `znak` z kolejnymi liczbami całkowitymi z tego przedziału jako jej argument:

```
znak(-2)
znak(-1)
znak(0)
znak(1)
znak(2)
```

```
-2 jest ujemna
-1 jest ujemna
0 to zero
1 jest dodatnia
2 jest dodatnia
```

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.

Jednym z typów pętli jest pętla **while**, która powoduje wykonywanie zawartego w niej kodu dopóki podany warunek jest spełniony. Do wykonania powyższego zadania służy pętla **while** w następującej postaci:

```
i = -2
while i <= 2:
  znak(i)
  i = i+1
```

```
-2 jest ujemna
-1 jest ujemna
0 to zero
1 jest dodatnia
2 jest dodatnia
```

Zadanie 2.2.1

Napisz funkcję, przyjmującą dwa argumenty a i b, która obliczy i zwróci sumę liczb całkowitych większych od a i mniejszych od b.

Zadanie 2.2.2

Napisz funkcję który wypisze liczby od 0 do 20 z pominięciem liczb podzielnych przez wartość określoną w jej argumencie.

3 Głębiej w Pythona (cz. 1)

3.1 Wielokrotne przypisanie

```
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 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.

3.2 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")
```