Języki programowania – Python
Własne funkcje projektanta
Obsługa błędów

Marzena Nowakowska

Wydział Zarządzania i Modelowania Komputerowego
Politechnika Świętokrzyska

Budynek C, p. 3.21

spimn@tu.kielce.pl

Funkcja w programie komputerowym

Funkcja - wyodrębniony, nazwany fragment kodu, zdefiniowany w celu jego wielokrotnego użytkowania w różnych miejscach programu komputerowego (funkcja jest więc swego rodzaju podprogramem).

Wynikiem działania **funkcji** może być:

- pojedyncza dana/wartość określonego typu,
- zbiór danych/wartości,
- realizacja jakiegoś zadania (np. modyfikacja wartości zmiennych, transfer do lub z pliku, komunikacja ze standardowymi strumieniami we/wy, działanie na pliku dyskowym),
- Nic, funkcja nie zwraca wartości (*None*)

Aby funkcja w programie zadziałała musi być w nim **wywołana**. Zanim funkcja zostanie wywołana, musi być w programie **wcześniej zdefiniowana**, aby interpreter Pythona jej znał treść.

Funkcja najczęściej pracuje na danych, które są zazwyczaj przekazywane przez **parametry**. Parametry mogą być narzędziem, za pomocą którego funkcja komunikuje się z programem, który ją wywołał.

Gotowe funkcje Pythona

- Funkcje **wbudowane**; dostarczane wraz z oprogramowaniem. Projektant wywołuje funkcje w programie, bez konieczności wcześniej ich podawania definicji.

  Przykłady: *input*, *print*, *type*, *max*, *min*, wszystkie funkcje rzutowania typów standardowych (np. *int*, *float*, *list*, *tuple*)

- Funkcje **modułów**; zdefiniowane w modułach tematycznych importowanych do programu (instrukcja *import*) przed wywołaniem funkcji. Projektant wywołuje funkcje w programie, ale wcześniej musi zaimportować z modułu potrzebną funkcję albo cały moduł.

  Przykłady:

  - import wybranej funkcji

    from math import sin; y= sin(45)

  - import modułu

    import random

    print(random.random()) # losowanie z przedziału [0, 1)

    print(random.randrange(101)) # losowanie z przedziału [0, 101)

Funkcje projektanta

- Funkcje **projektanta** (własne funkcje użytkownika). **Definiowanie funkcji** przez programistę musi być zgodne ze składnią:

  def ***nazwa_funkcji***(*lista_parametrów_**formalnych**_jeżeli_są*):

  treść funkcji (algorytm, ciało funkcji)

  return *jeżeli_jest_wyrażenie_którego_wartość_zwracana*

  Wiersz nagłówka *def* określa nazwę funkcji oraz listę parametrów formalnych (argumentów).

  Polecenie *return* może znajdować się w dowolnym miejscu ciała funkcji (uzasadnionym logiką algorytmu) i kończy wykonywanie wywołanej funkcji, zwracając sterowanie do miejsca w programie za miejscem wywołania.

  Jeżeli przy słowie *return* nie stoi żadne wyrażenie lub stoi słowo kluczowe *None* lub brak jest instrukcji *return*, to funkcja zwraca wartość *None* (nic).

- **Wywołanie funkcji** przez programistę musi być zgodne ze składnią:.

  nazwa_funkcji*(lista_parametrów_**aktualnych**_jeżeli_są)*

Funkcja projektanta zwracająca wynik – przykład

```
def maksimum(x, y):          # Definicja funkcji
    if x >= y:
        return x
    else:
        return y

# Wywołania funkcji dla różnych parametrów (i typów) aktualnych, które są stałymi
print("Większa z dwóch liczb: ", maksimum(3, 3))                    # → 3
print("Większy z dwóch tekstów: ", maksimum("alabama", "ala"))# → 'alabama'
print("Większy z dwóch tekstów: ", maksimum("Alabama", "ala"))# → 'ala'
print("Większa z dwóch list: ", maksimum([11,12,13], [14,15]))     # → [14, 15]
print("Błąd - niezgodność typów argumentów: ", maksimum("Alabama", 14))
# → TypeError (błąd typu danych) – BŁĄD WYKONANIA
# Wywołania funkcji dla różnych parametrów aktualnych, które są zmiennymi
zm1 = (20, 30, 40); zm2 = (20, -40, -50)
print("Większa z dwóch krotek: ", maksimum(zm1, zm2))          # → (20, 30, 40)
```

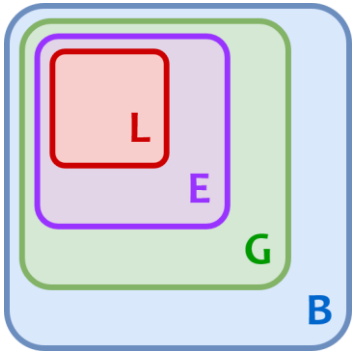
Funkcja projektanta zwracająca wynik liczbowy lub *None* - przykład

```
def dzielenie(liczba1, liczba2):
    if liczba2 == 0:
        return None
    else:
        return liczba1/liczba2

# Wywołanie funkcji
a = float(input('Podaj dzielną: '))
b = float(input('Podaj dzielnik: '))
wynik=dzielenie(a, b)
if wynik == None:
    print("Brak wyniku – dzielnik równy zero!")
else:
    print('Wynik dzielenia: %s' % wynik)
```

Przestrzenie nazw w Pythonie

Przestrzeń nazw to zbiór aktualnie zdefiniowanych nazw symbolicznych wraz z informacjami o obiekcie, do którego odwołuje się każda nazwa. Można myśleć o przestrzeni nazw jako o słowniku, w którym nazwy obiektów są kluczami, a same obiekty są wartościami słownika.



W Pythonie wyróżnia się cztery rodzaje przestrzeni nazw dla zmiennych:

Lokalne (***L****ocal*) określona dla bieżącej funkcji lub metody pewnej klasy (funkcji wywoływanej na rzecz obiektu). Zmienne tej przestrzeni (lokalne) są dostępne tylko w trakcie działania funkcji lub metody.

Otaczające (***E****nclosing*) dotyczy przypadku zagnieżdżenia funkcji (definiowanie jednej funkcji w innej funkcji). Odnosi się do przestrzeni nazw tej funkcji (nadrzędnej), w której inna funkcja (podrzędna) jest zagnieżdżona. Funkcja podrzędna może odwoływać się do zmiennych funkcji nadrzędnej.

Globalne (***G****lobal*) zawiera dowolne nazwy zdefiniowane na poziomie programu głównego. Powstaje po uruchomieniu głównego ciała programu i pozostaje do momentu zakończenia działania interpretera. Interpreter tworzy również globalną przestrzeń nazw dla każdego modułu, który jest ładowany do programu.

Wbudowane (***B****uilt-in*) zawiera nazwy wszystkich wbudowanych obiektów. Są one dostępne przez cały czas działania Pythona. Wykaz dostarcza polecenie: *dir(__builtins__)*.

Parametry funkcji
Zmienne lokalne funkcji

Parametr formalny to zmienna wymieniona w nawiasach w definicji funkcji; jeżeli funkcja ma więcej parametrów, to są one separowane przecinkami na liście parametrów. Parametry formalne przyjmują konkretne wartości w momencie wywołania funkcji. Te konkretne wartości są parametrami aktualnymi. Lista i typy (kontekst) parametrów formalnych i aktualnych w programie muszą się zgadzać.

Oprócz parametrów **funkcja może wykorzystywać swoje zmienne**, które są definiowane wewnątrz funkcji – to są zmienne lokalne.

Parametry formalne są zmiennymi (nazwami) lokalnymi dla podprogramu.

Zarówno parametry formalne jak i zmienne lokalne funkcji tworzą przestrzeń nazw lokalnych w momencie jej wywołania. Przestrzeń ta jest niszczona, gdy funkcja kończy swoje działanie i następuje z tej funkcji powrót w miejsce po jej wywołaniu.

Zmienne lokalne i globalne

```
x = 99 # zakres globalny
def funkcja1 (y):
    # zakres lokalny
    x = 100 # przesłanianie
    z =  x+y
    return z
def funkcja2 (y):
# zakres globalny; zmiana
# wartości zmiennej x globalnie
    global x
    x = 1000
    z = x+y
    return z
```

```
# Ciąg dalszy programu

print("Zmienna x przed wołaniem
funkcji1: ", x,
        "\nFunkcja 1: ", funkcja1(1),
        "\nZmienna x po wołaniu
funkcji1: ", x, "\n")
print("Zmienna x przed wołaniem
funkcji2: ", x,
        "\nfunkcja 2: ", funkcja2(1),
        "\nZmienna x po wołaniu
funkcji2: ", x)
```

Zmienne lokalne i otaczające

```
# Zad_10-funkcje_0. Funkcja wyznacza listę zawierająca ilorazy
# kolejnych elementów listy liczniki i liczby mianownik
def lista_ilorazow(liczniki, mianownik):
  def dzielenie(liczba):
        print("mianownik: ", mianownik)
        if mianownik == 0:
           return None
        else:
           return liczba/mianownik

  ilorazy=[]
  for el in liczniki:
     ilorazy.append(dzielenie(el))
  return ilorazy

#  Program główny
lst_arg=[1,2,3,4,5,6,7,8]
mian=3
wyniki=lista_ilorazow(lst_arg, mian)
```

Mutowalność

Obiekt **niemutowalny** (niezmienny) to taki, której wartości nie można zmienić „w miejscu” jej położenia (na stosie). Należą do nich: **liczby**, **wartości logiczne**, **łańcuchy**, **zakresy (*range*)** i **krotki**.

To oznacza, że po utworzeniu obiektu nie można zmienić jego zawartości po utworzeniu. Można natomiast do zmiennej identyfikującej taki obiekt przypisać inną wartość.

Przykład 1.

```
krotka = (0, 1, 2, 3);        krotka[0] = 4      # nieprawidłowa operacja
krotka = (4, 1, 2, 3)                            # prawidłowa operacja
```

Dane prawie wszystkich innych typów są mutowalne. Obiekt **mutowalny** (zmienny) to taka, której wartość można zmienić „w miejscu” jej położenia (na stosie). Są to np.: **listy**, **słowniki**, **zbiory**, **obiekty** zdefiniowane przez projektanta (programistę).

To oznacza, że po utworzeniu obiektu można zmienić jego zawartości po utworzeniu. Można również do zmiennej identyfikującej taki obiekt przypisać inną wartość.

Przykład 2.

```
lista = [0, 1, 2, 3];   lista[0] = 4      # prawidłowa operacja
lista = [4, 1, 2, 3]                      # prawidłowa operacja
```

Obiekty niemutowalne i mutowalne jako parametry funkcji – demo w Spyder

```
def dodawanie_3_obiektow(ob1, ob2, ob3):
   """ Wszystkie parametry funkcji muszą być tego samego typu """
   wynik1=ob1+ob2+ob3
   typ=type(ob1)
   print(typ)
 #   ob1+= (10 if typ == int else ("10" if typ == str else [10]))
   if (typ == int) | (typ == float):                    # NIEMUTOWALNY
     ob1+=10;    ob2=100
   elif typ==str:                                       # NIEMUTOWALNY
     ob1+="==";     ob2="MIAŁA"
   elif typ == list:                                    # MUTOWALNY
    ob1+=[10];    ob2=[1000]
   else:
    pass
   print("%s %s %s" % (ob1, ob2, ob3))
   wynik2=ob1+ob2+ob3
   return wynik1, wynik2                                # funkcja zwraca wartości dwóch obiektów

arg1=22.2; arg2= 1000 ; arg3= 10000
# arg1="aa"; arg2="bb"; arg3="cc"
# arg1=["xx"]; arg2=[12, 14]; arg3=["yy", 44]
a,b=dodawanie_3_obiektow(arg1,arg2,arg3)
```

Parametry opcjonalne funkcji

W Pythonie istnieje możliwość zdefiniowania **parametrów opcjonalnych**. W takiej sytuacji w definicji funkcji **parametr musi mieć nazwę i przypisaną mu wartość**, która ma być **wartością domyślną**. Jeżeli nie wszystkie parametry funkcji są opcjonalne, to wartości domyślne mogą mieć tylko parametry na końcu listy parametrów.

```
def komunikat(tresc = "Programowanie", ile = 1):
    print("Wywołanie funkcji dla parametrów: ", tresc, ", ", ile)
    print("Wyswietlane: ", tresc * ile,"\n")
    return

# Jakie są wyniki poniższych wywołań funkcji
komunikat(3, 3)
komunikat()
komunikat("Hello ", 5)
komunikat("Hello")  #nieprawidłowe: komunikat(, 3)
komunikat(tresc="-Hello-", ile=4)
komunikat(ile=4, tresc="-Hello-")
komunikat(ile=3)
```

Funkcja/wyrażenie lambda

Funkcja lambda to mała anonimowa (nienazwana) funkcja. Funkcja lambda może przyjmować dowolną liczbę argumentów, ale może mieć tylko jedno wyrażenie. Składani funkcji jest następująca:

lambda *lista_argumentów* : *wyrażenie*

Funkcje *lambda* zapewniają wygodny sposób na pisanie skróconych wyrażeń i utrzymywanie porządku bez zbędnego tworzenia wielu nazwanych funkcji. Jednak nadużywanie funkcji *lambda* może utrudnić odczytywanie kodu, dlatego należy z tego narzędzia korzystać z umiarem.

Przykład

```
# Przypisanie funkcji do zmiennej, a następnie wykonanie jej
zmienna = lambda x,y: x+y
print(zmienna(2,3))
# Wywołanie funkcji bez przypisywania do zmiennej
print((lambda x,y: x+y)(3,4))
```

Rodzaje błędów i wyjątki w programie

Błędy składni (syntaktyczne) powstają, gdy parser (program wykonujący analizę składniową) Pythona nie jest w stanie zrozumieć wiersza kodu. Błędy składniowe są prawie zawsze krytyczne, tj. prawie nigdy nie ma sposobu na pomyślne wykonanie fragmentu kodu zawierającego błędy składniowe. Większość błędów składniowych to literówki, nieprawidłowe wcięcia lub nieprawidłowe argumenty.

Błąd wykonywania (wyjątki, sytuacje wyjątkowe) występuje, gdy program jest poprawny składniowo, ale zawiera problem, który jest wykrywany tylko podczas wykonywania programu**.** Powoduje przerwanie wykonywania programu. Kod jest zrozumiały (nie ma błędów składni), ale problematyczny fragmenty kodu jest niemożliwy do wykonania. Przykłady: dzielenie przez zero, odczyt nieistniejącego pliku, próba dostępu do danej poza zakresem (element listy o numerze większym niż długość listy).

Błąd logiczny występuje, gdy program działa, ale może działać nieprawidłowo i generować błędne wyniki (niezgodne z oczekiwaniami) lub wykonywać się niezgodnie z oczekiwaniami lub planami projektanta (program wykonuje to, co ma napisane a nie to, co myśli lub chce programista). Przykład: nieskończenie działająca pętla.

Błędy logiczne można wykryć wykorzystując debugger.

Śledzenie programu

Debugger to program komputerowy służący do dynamicznej analizy innych programów w celu odnalezienia i identyfikacji zawartych w nich błędów (w szczególności logicznych), zwanych z angielskiego bugami.

Debugger umożliwia śledzenie działania programu, wykonując ten program **krok po kroku** (funkcjonalność większości IDE). Debuggery mają także możliwość śledzenia tego, jak zmienia się zawartość zmiennych (dane), lokalnych i globalnych.

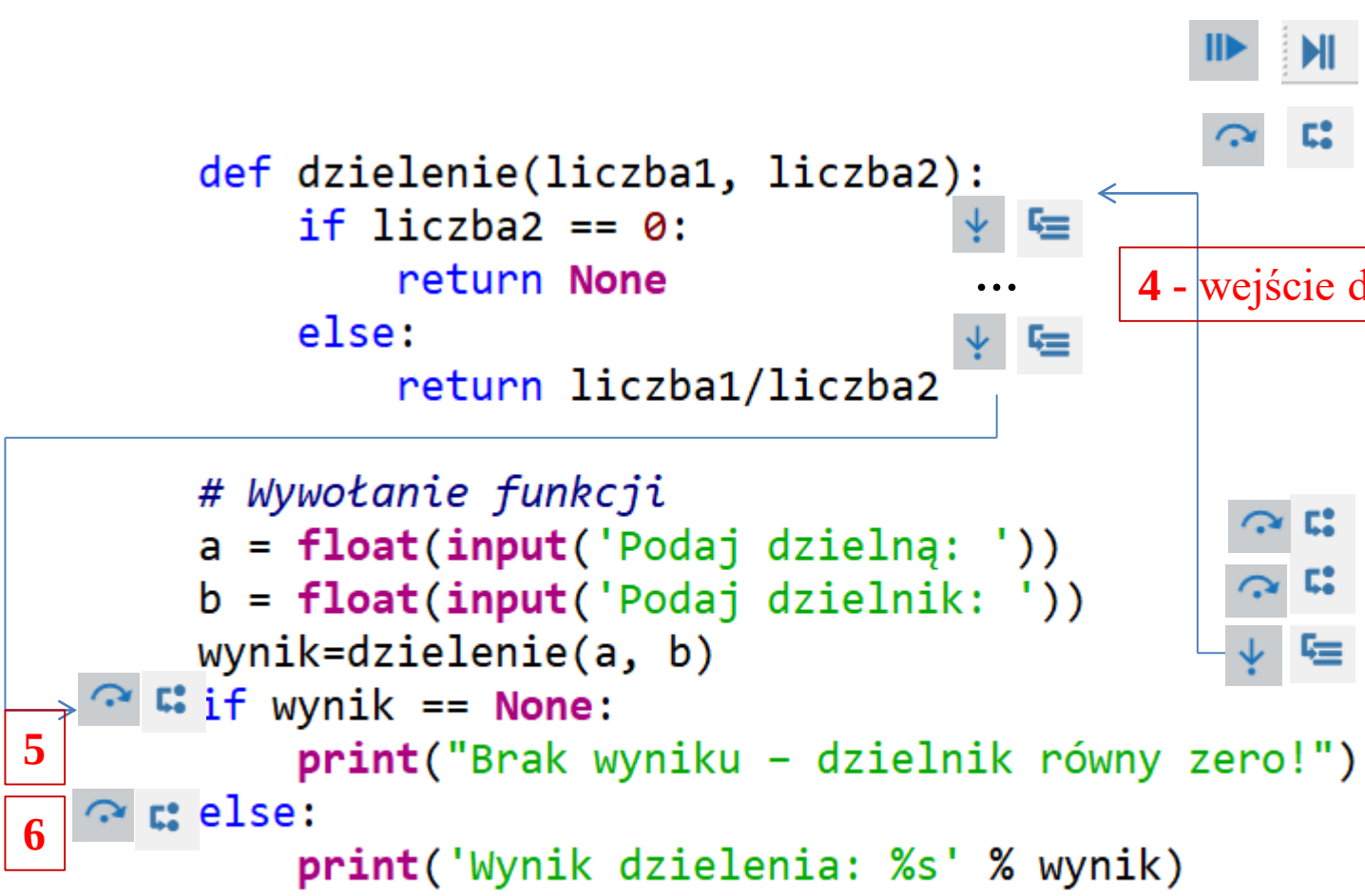
Proces nadzorowania wykonania programu za pomocą debuggera określa się mianem **debugowania**.

Aby sprawdzić program w określonym punkcie należy w wybranej linii wstawić **punkt przerwania** (*breakpoint*). Wtedy po uruchomieniu śledzenia, program wykonuje się niekrokowo do punktu przerwania. W tym momencie można sprawdzić wartości ziemnych w miejscu programu, w którym punkt jest ustawiony.

W środowisku *spyder* po włączeniu śledzenia zachęta w oknie konsoli ma postać: ipdb>. Jednocześnie w oknie edytora kursor strzałki ustawia się na aktualnie wykonywanej instrukcji.

Powrót do standardowej linii poleceń – po wyłączeniu śledzenia.

Śledzenie programu – obsługa





Przyciski umieszczone ww. obok siebie realizują te same operacje w debuggingu, są jednak inaczej oznaczone w różnych wersjach Spydera.

Demo: Zad_13-śledzenie.py

Obsługa wyjątków

Moment zajścia błędu wykonania jest sygnalizowany jako **sytuacja wyjątkowa** (nieoczekiwana) – określa się to jako podniesienie (zgłoszenie) wyjątku (*raise exception*). Większość wyjątków nie jest obsługiwana przez program i powoduje wyświetlanie komunikatu o błędzie, np.:

- *KeyError* (próba wykorzystania nieistniejącego klucza w słowniku),
- *NameError* (odwołanie do nieistniejącej zmiennej),
- *ZeroDivisionError* (dzielenie przez zero).

Standardowe nazwy wyjątków są wbudowanymi mnemonicznymi identyfikatorami. Można je wykorzystać przy pisaniu programu, który obsługuje wybrane wyjątki.

Projektant może obsłużyć wybrane wyjątki za pomocą instrukcji ***try… except***. Krytyczny fragment kodu, w którym może wystąpić błąd jest umieszczony w części *try*. Kod, który obsługuje wyjątek lub wyjątki jest zapisany w bloku *except*. Po klauzuli *try* musi wystąpić co najmniej jedna klauzula *except*. Liczba bloków *except* zależy od decyzji projektanta. Instrukcję *try* można zagnieżdżać.

Wykaz wyjątków w Pythonie można znaleźć pod adresem:
https://www.w3schools.com/python/python_ref_exceptions.asp

Składnia obsługi wyjątków

try:

blok kodu, w którym mogą wystąpić błędy

except *NazwaBłędu1*:

blok obsługi błędu o podanej nazwie

except (*NazwaBłędu2*, *NazwaBłędu3*):

blok obsługi błędów o podanych nazwach (opcja)

except:

blok w przypadku błędów niewyszczególnionych wcześniej (opcja)

else:

blok kodu, który ma być uruchomiony, gdy nie ma błędu w bloku try *(opcja)*

finally:

blok kodu wykonywany po przejściu wszystkich wcześniejszych bloków obsługi błędów

```
try:
    print("\nObliczenie wielkosci zmiany ceny towaru")
    cena_przed=float(input("Podaj cenę towaru przed zmianą: "))
    cena_po=float(input("Podaj cenę towaru po zmianie: "))
    zmiana= -(cena_przed-cena_po)/cena_przed*100
except ValueError:
    print('\nWprowadzony ciąg znaków niezgodny z formatem liczby. Brak wyniku')
    zmiana=None
except ZeroDivisionError:
    print('\nWprowadzona cena przed zmiana nie może być zerem. Brak wyniku')
    zmiana=None
except:
    print("\nInny błąd w przetwarzaniu danych. Brak wyniku")   #np. Ctrl+c (przerwanie operacji)
    zmiana=None
else:
    print('\nBrak błędów przetwarzaniu danych. Wynik poprawny')
finally:
    print('\nProcent zmiany ceny: ' , end='')
    if zmiana == None:
        print(zmiana)
    else:
        print("%+6.2f%s" %(zmiana, "%"))
```

Obsługa wyjątków.
Przykład Zad_33-exception-0.py