

Funkcje (w matematyce)

W matematyce: dwa sposoby patrzenia na funkcje.

1. Jako przyporządkowanie (z odpowiednią dziedziną i przeciwdziedziną).

$$\begin{aligned} f: \mathbb{R} &\rightarrow \mathbb{R}, & f(x) &= x^2, \\ g: \mathbb{R}^2 &\rightarrow \{-1, 0, 1\}, & g(x, y) &= \operatorname{sgn}(x \cdot y). \end{aligned}$$

Wyrażenie $f(2)$ jest tożsame z jego wartością, 4.

2. Jako obiekt matematyczny, na którym można operować.

$$\begin{aligned} T: \mathbb{R}^{\mathbb{R}} &\rightarrow \mathbb{R}, & T(h) &= h(3), \\ S: \mathbb{R}^{\mathbb{R}} \times \mathbb{R}^{\mathbb{R}} &\rightarrow \mathbb{R}^{\mathbb{R}}, & S(f_1, f_2) &= f_1 + f_2. \end{aligned}$$

$$[(f_1 + f_2)(x) := f_1(x) + f_2(x)]$$

Funkcje (w Pythonie)

Funkcje w Pythonie (i wielu językach programowania): coś innego.

Funkcje to podprogramy, które:

- Można uruchomić („wywołać”), podając im argumenty.
- Zwracają wartość.
- Mogą wykonywać instrukcje niezwiązane z wyliczaniem tej wartości.

Funkcje są też obiektami – o tym później.

Składnia definicji funkcji (pierwsza wersja):

```
nazwa parametry[formalne]  
def fun (  $x_1, x_1, \dots, x_n$  ) :  
    blok instrukcji } treść
```

I składnia wywołania, dla $x_i = a_i$, gdzie a_i to konkretne obiekty:

```
argumenty[faktyczne]  
fun (  $a_1, a_1, \dots, a_n$  )
```

Przykład (nieuproszczony).

```
def f(x):  
    r = x ** 2  
    return r  
  
y = f(2)
```

`def f(x):` i blok poniżej: definiuje funkcję: nazwę, parametry formalne, instrukcje.
`f(2)` – wyrażenie (tzn. napis posiadający wartość). Wywołuje `f` z argumentem `x=2`.

`return expr` – instrukcja, która przerywa działanie funkcji. Wartość opcjonalnego wyrażenia `expr` jest wartością przyjmowaną przez wyrażenie, które wywołało funkcję.

Dowolnie wiele return:

```
def g(x, y):  
    z = x * y  
    if z > 0:  
        return 1  
    elif z < 0:  
        return -1  
    return 0
```

Domyślnie, return zwraca None (tj. „return” = „return None”). Jeśli w treści funkcji skończą się instrukcje do wykonania, i nie zostanie napotkane return, wywołanie funkcji zwraca None.

```
def print_squares(n,m):  
    print(f" {n}^2 = {n ** 2}")  
    print(f" {m}^2 = {m ** 2}")
```

Funkcje (w Pythonie)

Pewne zalety funkcji:

- Modularyzacja kodu: funkcja to „czarna skrzynka”, której możemy powierzyć wykonanie pewnych instrukcji lub wyliczenie wartości wyrażenia.
- Wielokrotne użycie tego samego kodu: jeśli program ma wiele fragmentów różniących się szczegółami, można zrobić z nich jedną funkcję, gdzie „szczęgół” jest parametrem.
- Lepsza organizacja kodu: funkcje mają wyspecjalizowane zadania i konkretne znaczenie.
- Ukrycie szczegółów implementacji.

Np.

```
def is_prime(n): # True/False
    ...

...
if is_prime(a) and is_prime(b): ...
```

Zasięg nazw (scope of variables): „gdzie widać nazwę i do czego się ona odnosi”.

Ramka: kontekst wykonywania instrukcji w Pythonie.

- Ramka globalna – zawsze obecna.
- Ramka wywołania funkcji – tworzona dla konkretnego wywołania.

Nazwy (i informacja, jakie obiekty nazywają) przynależą do ramki.

- Nazwy utworzone poza wywołaniem funkcji należą do ramki globalnej.
- Nazwy utworzone w wywołanej funkcji (w tym parametry formalne) należą do ramki tego wywołania.
- Nazwy w różnych ramkach mogą się powtarzać, lecz nazywać inne obiekty.

Odnosząc się do nazw: najpierw sprawdzana jest ramka aktualnego wywołania funkcji (jeśli istnieje), następnie ramka globalna.

Funkcje jako obiekty

Funkcja jako obiekt: użyta np. jako argument innej funkcji.

```
def call_twice(f, x):  
    f(x)  
    f(x)  
  
call_twice(print, "Hello world!")
```

```
def square(x):  
    return x ** 2  
  
def add(x):  
    return x + 1  
  
def compose(f, g):  
    def h(x):  
        return f(g(x))  
    return h
```

```
f1 = compose(square, add) # zlozenie funkcji
```