Formatowanie napisów

Proste wypisywanie kilku obiektów (ich napisowych reprezentacji):

```
print("Liczba", n, "jest pierwsza.")
```

Cel: uzyskać pojedynczy napis postaci "Liczba n jest pierwsza", gdzie n jest parametrem; oraz kontrolować sposób formatowania n.

Trzy (cztery) sposoby formatowania napisów w Pythonie:

- „Stary" wzorowany na C.
- „Nowy" poprzez metody typu str, w stylu Javy/C#.
- „Nowszy" przez *interpolowane napisy* (f-stringi) – ten omówimy.
- „Najnowszy" (Python 3.14) – *template strings*.

f-stringi

f-stringi (formatted string) - wyrażenie wyliczające się do napisu (obiektu typu str).
Przykład:

```
n = 13
s = f"Liczba {n} jest pierwsza." # "Liczba 13 jest pierwsza."
...
print(s)
```

- Dowolnie wiele „formatek" postaci "{...}" zastąpionych odpowiednimi napisami.
- Formatki są postaci "{expression}" lub "{expression:options}".

```
a, b = 2, 3
s = "napisem"

f"jestem {s}em" # "jestem napisem"
f"Suma {a} i {b} to {a+b}." # "Suma 2 i 3 to 5."
f"Wartośc {a=}" # "Wartość a=2"
f"Jestem {s}em, który jest fajnym {s}em. A poza tym {b=}."
```

f-stringi

Kilka przykładów na opcje (ogólnie jest ich dużo więcej):

```
a, b = 2, 3
s = "napis"
x = 3.5678 # float

f"{s:20}"      # "napis               "
f"{s:>20}"     # "               napis"
f"{s:^20}"     # "       napis        "
f"{s:x^20}"    # "xxxxxxxnapisxxxxxxxx"

f"{a:^20}"     # "          2         "

f"{x:.4}"      # "3.568"
f"{x:.2}"      # "3.6"
f"{x:10.4}"    # "      3.568"

f"{x:ś^10.4}" # "śś3.568śśś"
```

Funkcje

Funkcje (intuicyjnie) to podprogramy, które:

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

Składnia definicji funkcji (minimalna wersja):

def $\overbrace{\text{fun}}^{\textit{nazwa}}(\overbrace{x_1, x_1, \ldots, x_n}^{\textit{parametry [formalne]}}):$
blok instrukcji } treść

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

fun($\overbrace{a_1, a_1, \ldots, a_n}^{\textit{argumenty [faktyczne]}}$)

Przykłady

```
def square(x):
    y = x ** 2
    return y # "zwróć y"

a = square(50) # 2500
b = square(2) + square(3) # 2**2 + 3**2 == 13
print(f'Kwadratem {b} jest {square(b)}')
```

Krócej:

```
def square(x):
    return x ** 2 # return expression
```

Treść funkcji – dowolne* instrukcje, dowolnie wiele return:

```
def absolute(x): # wartość bezwzględna, wbudowany odpowiednik: abs()
    if x >= 0:
        return x
    return -x # wykona się tylko wtedy, gdy x < 0
```

Przykłady

Blok to dowolne instrukcje: funkcja może powodować efekty uboczne (niezwiązane z wyznaczaniem wyniku). Celem wywołania funkcji mogą być tylko te efekty:

```
def pprint(s):
    print("☺☺☺")
    print(f"☺☺☺{s}☺☺☺")
    print("☺☺☺")
```

Koniec instrukcji równoważny jest z napotkaniem return, a samo return jest równoważne z return None.

```
pprint("Hurra")

☺☺☺
☺☺☺Hurra☺☺☺
☺☺☺
```

Znane już input() oraz print() to wbudowane funkcje.

Przykłady

Test na pierwszość (wciąż „matematycznie prymitywny") jako funkcja:

```
def is_prime(n):
    if n < 2:
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True
```

```
a = 13
b = 10
if not is_prime(a) or not is_prime(b):
    print(f"{a} lub {b} nie jest pierwsza.")
```

Przykłady

Parametry funkcji są przywiązane do konkretnego wywołania konkretnej funkcji:

```
def f(x):
    return x + 1

def g(x):
    return 2 * f(x)

print(f(10)) # 11
print(g(10)) # g(10) == 2*f(10) == 2 * (10+1)
```

Przykłady

Rozważmy rekurencyjnie zdefiniowany ciąg liczb naturalnych:

$$a_0 = 0, \text{ oraz } a_n = 2a_{n-1} + 1 \text{ dla wszystkich } n > 0.$$

Zatem

$$\begin{aligned} a_1 &= 2a_0 + 1 &&= 1, \\ a_2 &= 2a_1 + 1 &&= 3, \\ a_3 &= 2a_2 + 1 &&= 7, \\ &\dots \end{aligned}$$

(mamy ogólnie $a_n = 2^n - 1$, ale zakładamy, że tego nie wiemy)
Rekurencyjna implementacja w Pythonie – prawie z definicji:

```
def a(n):
    if n == 0:
        return 0
    return 2 * a(n - 1) + 1
```

Listy

Listy – obiekty typu list, reprezentujące ciągi obiektów. Podobne do krotek, ale *zmienialne*. Konstrukcja listy „ręcznie" podobna jak dla krotek:

```
tup = (1, "abc", 3.0) # krotka, można też pominąć nawiasy
lst = [1, "abc", 3.0] # lista, nawiasy konieczne
```

Konstrukcja z innego ciągu:

```
lst = list("abcde") # ["a", "b", "c", "d", "e"]
lst = list(range(2, 10, 2)) # [2, 4, 6, 8]
```

Krotki (tuple), listy (lst) oraz napisy (str) to ciągi i mają pewne wspólne operacje (i wiele z nich działa też dla range):

```
s = "abcde"
t = ('b', 'a')
lst = [2.0, 1, 4]
len(s) # długość napisu (5)
len(t) # 2
len(lst) # 3
```

Operacje na ciągach

Kilka dalszych przykładów:

```
s = "abcde"
t = ('b', 'a')
lst = [2.0, 1, 4]

max(lst) # 4
max(t) # "b"
max(s) # "e"

min(lst) # 1

sum(lst) # 7.0

t + t #  ('b', 'a', 'b', 'a')
lst + lst # [2.0, 1, 4, 2.0, 1, 4]
```

Operacje na ciągach

Iteracja:

```
for c in s:
    ... # po znakach
for x in t:
    ... # po elementach
for x in lst:
    ... # po elementach
```

Indeksowanie: $\mathrm{sequence}[\mathrm{index}]$ – element o indeksie index (licząc od 0):

```
lst[0] # 2.0
lst[2] # 4

"Niedźwiedź"[4] # "ź"

(t+t)[2] # "b"
```

Operacje na ciągach

(Extended) slicing (wycinanie):

```
lst = [0,1,2,3,4,5,6,7,8,9,10]
lst[4:8] # [4, 5, 6, 7]

"abcde"[1:3] # "bc"

"abcde"[1:] # "bcde"
"abcde"[:3] # "abc"

"abcde"[:] # "abcde"

lst[2:7:2] # [2, 4, 6] -- indeksy jak dla range(2, 7, 2)
```

Tylko dla list (i to odróżnia je od krotek) – podmiana elementów:

```
lst = [1, 2, 3]
lst[1] = ":)"
print(lst) # [1, ":)", 3]
```