Informacje o wykładzie

- *Programowanie 1 R (Python)*
- Założenia: znajomość elementarnego Pythona lub zdolność szybkiego opanowania podstaw.
  - Podstawowe wyrażenia i instrukcje.
  - Instrukcje warunkowe, proste pętle.
  - Pisanie prostych funkcji.
  - Znajomość środowiska.
- Strona wykładu (uwaga na `_r` w url!):

  `https://math.uni.wroc.pl/~jagiella/p1python_r/`

  (lista omówionych zagadnień, materiały z wykładu, przydatne linki)
- Strona ćwiczeń/laboratoriów (uwaga na `_r` w url!):

  `https://math.uni.wroc.pl/~jagiella/p1python_r/lab/`

  (listy zadań, przykładowe rozwiązania)
- Konsultacje: TBA.

Strony formalne i zaliczenie

- Wykład kończy się praktycznym (pisemnym) egzaminem.
- Dopuszczenie do egzaminu: zaliczenie laboratoriów przez wykonywanie list zadań.
- Listy zadań mają charakter zadania domowego i należy je wykonywać samodzielnie.
- Listy zadań będą pojawiać się po każdym wykładzie i obowiązują (domyślnie) na następny tydzień.
- Prowadzący mogą modyfikować harmonogram w swojej grupie lub stosować własne listy*.
- 50% na 3.0, 60% na 3.5, ..., 90% na 5.0.

Przypomnienie i uzupełnienie podstaw Pythona

Plan na początek: niuanse związane z podstawowymi elementami Pythona.

- Kolejność operacji w prostych wyrażeniach: arytmetyka → komparatory → logika.
- Nazwy to etykiety (aliasy) obiektów, a nie obiekty.
- Tworzenie obiektów i (pozorna) konwersja typów.
- Obiekty iterowalne w pętlach.
- Funkcje: parametry pozycyjne vs. parametry nazwane.
- „Wszystko jest obiektem", w tym funkcje i typy obiektów.
- Arytmetyka zmiennoprzecinkowa, dokładność reprezentacji.
- Drugi wykład: formatowanie napisów (sposób „stary", „nowy" i „nowszy").

Kolejność operacji

Wyrażenia w Pythonie: wyliczają się do wartości, co do zasady nie produkują efektów ubocznych (nie zmieniają stanu programu). Przykładowo:

- stałe liczbowe i napisowe,
- nazwy,
- wyrażenia postaci `wyrażenie1` $\circ$ `wyrażenie2`, gdzie $\circ$ to operator binarny (`+`, `-`, `*`, `/`, `<=`, `<` , `and`, `or`, ...),
- indeksowanie: `wyrażenie1[wyrażenie2]`,
- wywołania funkcji i metod obiektów,
- ...

Koncentrujemy się teraz na wyrażeniach ze stałymi, operatorami (arytmetycznymi, porównywania i logicznymi) i nawiasami.

Kolejność operacji

O kolejności wykonywania operatorów decydują nawiasy, priorytety operatorów i kolejność występowania. Kolejność wykonywania co ważniejszych operatorów:

- `**`
- jednoargumentowy +, − (jak w +5 i −a)
- *, /, %, // (operacje multyplikatywne)
- dwuargumentowe +, − (jak w a+b)
- <=, <, >=, >, ==, !=, `in`, `not in` (porównania, należenie)
- `not`
- `and`
- `or`

Mamy zatem grupy operacji: arytmetyka, porównania (komparacja), logika.
W przykładowym (dość naturalnym) wyrażeniu

$$x + 1 > 5 * y \text{ and } 2 * y < 6$$

nie trzeba żadnego nawiasu.

Nazwy

Nazwa nie jest tożsama z obiektem.

```
x = 1
lst = [1, 2, 3]
```

Analogia: nazwa to etykietka, przypisanie przykleja ją do obiektu (i nie tworzy kopii tego obiektu!). Wtedy nazwy (tutaj: `x` i `lst`) stają się wyrażeniami, których wartości to (nazywane przez nie) obiekty. Etykietkę można przekleić na inny obiekt (w tym obiekt innego typu):

```
x = "abc"
```

Ten sam obiekt może mieć jednocześnie wiele nazw:

```
lst2 = lst
```

Wtedy `lst` i `lst2` nazywają tę samą listę (przykład omówiony w `https://pythontutor.com/visualize.html`).

Tworzenie obiektów, pozorna konwersja typów

Niech `t` to nazwa typu, wbudowanego (np. `int`, `bool`, `list`, ...) lub nie (np. `datetime.date`). Wtedy

```
t()
t(x)
t(x, y)
t(...) # gdzie ... to ciąg wyrażeń oddzielonych przecinkami
```

są *wyrażeniami*, których wyliczenie tworzy nowy obiekt typu `t` (i obiekt ten jest staje się wartością tego wyrażenia). Tworzenie (= konstrukcja) obiektu typu `t` to inaczej *instancjacja* typu `t` (utworzony obiekt jest *instancją* typu `t`).
Konstrukcja obiektu zazwyczaj używa parametrów (powyżej: `x`, `y`, etc. to wyrażenia, których wynikami są obiekty, które stają się parametrami konstrukcji). Przykładowo

```
int()
int("125")
list("Hello world!")
datetime.date(1, 1, 1)
```

Tworzenie obiektów, pozorna konwersja typów

Wyrażenia te zachowują się jak każde inne wyrażenie.

```
x = int(2.5) + 5 # 7
s = str(123) + str(456) # "123456"
lst = list("abc") + list(range(10)) # ["a", "b", "c", 0, 1, ...]
lst[5] # 2

# indeksowanie wyliczonej na bieżąco listy
(list("abc") + list(range(10)))[5] # 2
```

Obserwacja (z ciekawymi konsekwencjami): instancjacja typu ma identyczną składnię, co wywołanie funkcji.

Podstawowe typy Pythona da się instancjonować z obiektów „podobnych" typów, np.

```
int(2.5), int("10"), float("2.5")
list("xyz"), list(range(10))
int(input("podaj x: "))
```

Pozornie przypomina to (znaną z niektórych języków) *jawną konwersję typów*, jednak formalnie wszystkie te wyrażenia tworzą nowe obiekty.

Obiekty iterowalne

Obiekt iterowalny – obiekt reprezentujący ciąg obiektów.
Przykład: napisy, listy, krotki (`tuple`), `range`. Zazwyczaj dostępne operacja: długość, indeksowanie.

```
a = "abc"
b = [1, 2, 3, 4] # lub (1, 2, 3, 4) dla krotki
c = range(10, 16) # ciąg 10, 11, 12, 13, 14, 15
d = range(10, 16, 2) # ciąg 10, 12, 14
```

```
len(a), len(b), len(c), len(d) # 3, 4, 6, 3
```

```
a[1] # "b"
b[2] # 3
c[2] # 12
d[2] # 14
```

Obiekty iterowalne

Wspólna i definiująca cecha obiektów iterowalnych: można po nich iterować (przebiegać po ich elementach).
Iteracja ma wiele form. Pętla `for`, ogólniejsza składnia:

```
for name in expression:
    ...
```

`name` – nazwa*, `expression` – wyrażenie, którego wartością jest obiekt iterowalny.

```
for i in range(10, 16):
    print(i)
```

```
for x in [1, 10, 100]:
    print(x)
```

```
for c in "napis" + "też napis":
    print(x)
```

Obiekty iterowalne

Inny przykład iteracji: w listach składanych (szerzej na nastepnym wykładzie):

```
lst = [x**2 for x in range(10)]
lst2 = [2*c for c in "napis"]
lst3 = [x - 1  for x in lst + lst]
```

Konstrukcja listy z dowolnego* obiektu iterowalnego – po prostu instancjacja listy:

```
list("napis")
list(range(10))
list("Hello" + " world!")
```

Funkcje: parametry pozycyjne i nazwane

Elementarna składnia definicji funkcji:

def $\overbrace{\text{fun}}^{\textit{nazwa}}$ ($\overbrace{\text{x}_1, \text{x}_2, \ldots, \text{x}_\text{n}}^{\textit{parametry [formalne]}}$):
blok instrukcji } treść

gdzie x_i to nazwy.

Składnia wywołania, dla $\text{x}_\text{i} = \text{e}_\text{i}$, gdzie e_i to wyrażenia:

fun ($\overbrace{\text{e}_1, \text{e}_2, \ldots, \text{e}_\text{n}}^{\textit{argumenty [faktyczne]}}$)

Funkcje: parametry pozycyjne i nazwane

Poszerzenie składni funkcji przez opcjonalne *argumenty nazwane*

def $\overbrace{\text{fun}}^{\text{nazwa}}$ ($\overbrace{\text{x}_1, \text{x}_1, \ldots, \text{x}_\text{n}}^{\text{parametry pozycyjne}}$, $\overbrace{\text{y}_1 = \text{d}_1, \text{y}_2 = \text{d}_2, \ldots, \text{y}_\text{m} = \text{d}_\text{m}}^{\text{parametry nazwane}}$):
blok instrukcji } treść

gdzie x_i, x_j to nazwy, a d_k to wyrażenia (oznaczające domyślne wartości parametrów).

Przy wywołaniu argumenty przekazywane są od lewej do prawej, mogą być też przekazywane przez nazwę. Parametry nazwane nie muszą zostać przekazane i przyjmą wartości domyślne. Przykłady:

```
def f(a, b, c=3, d=4):
    ...

f(10, 20)       # a=10, b=20, c=3, d=4
f(10, 20, 30) # a=10, b=20, c=30, d=4
f(a=10, b=20, d=30, c=40) # a=10, b=20, c=40, d=30
f(b=1, a=1)
```

„Wszystko jest obiektem”

Funkcje w Pythonie to tzw. „obywatele pierwszej kategorii” – są obiektami. Definicja funkcji postaci

```
def f(...):
    ...
```

w istocie tworzy obiekt będący funkcją oraz nadaje mu nazwę `f`. Można zatem:

```
g = f # nadać temu obiektowi nową nazwę
lst = [1, f, 10] # umieścić ten obiekt na liście
print(f) # zbadać jego napisową reprezentację (podobnie też: str(f))
```

Szczególna operacja, którą można wykonać na obiekcie, który jest funkcją: wywołanie tego obiektu.

```
f(...)
```

Funkcje to *obiekty wywoływalne*.

„Wszystko jest obiektem”

Typy obiektów też są obiektami. `int`, `float`, `list`, ... to nazwy konkretnych obiektów.

```
t = int
print(int)
lst = [int, float, int, list, bool, bool]
```

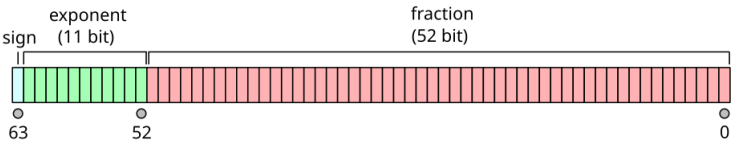
Podobnie jak funkcje, typy są wywoływalne – wywołanie typu (z odpowiednimi parametrami) to dokładnie instancjacja typu.

```
int(2.5)
float("3.14")
list("abc")
```

```
for t in [int, float, bool, list]:
    x = t()
    print(x)
```

Arytmetyka zmiennoprzecinkowa

Dwa fundamentalne typy obiektów reprezentujących liczby: `int` (liczby całkowite, teoretycznie dowolne) i `float`. `float` służy do reprezentowania liczb rzeczywistych, ale z ograniczeniami: tzw. liczby zmiennoprzecinkowe w standardzie IEEE podwójnej precyzji (64 bity).



Rysunek: https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

s – znak (1 bit).
E – wykładnik (liczba całkowita złożona z 11 bitów).
$b_{51}, b_{50}, \ldots, b_0$ - znaczące cyfry dwójkowe (52 bity).
Wartość reprezentowana przez 64 bity: $(-1)^s(1.b_{51}b_{50}b_{49}\ldots b_0)\cdot 2^{E-1023}$.

Arytmetyka zmiennoprzecinkowa

W praktyce dla liczb typu `float`:

- Zakres zaprezentacji: $\pm 1.7 \cdot 10^{308}$, ale liczby reprezentowane są w przybliżeniu.
- Dokładność do 15-16 znaczących cyfr dziesiętnych ($52 \log_{10} 2 \approx 15.65$).
- Obliczenia arytmetyczne wykonywane z (najlepszym możliwym) przybliżeniem.