UNIWERSYTET PEDAGOGICZNY
im. Komisji Edukacji Narodowej
w Krakowie
Instytut Informatyki

ROMAN CZAPLA

PROGRAMOWANIE OBIEKTOWE Z WYKORZYSTANIEM JĘZYKA PYTHON 3.x

wykład nr I

Kraków 03.03.2021

Obiekty w Pythonie

- **Obiekt** posiada następujące cztery cechy:
  - *tożsamość* - odróżnia go od innych obiektów;
  - *typ* - może być niezmienny lub zmienny (mutowalny);
  - *stan obiektu* - aktualna wartość przypisana do obiektu - obiekt może być w jednym lub w wielu stanach (jednak nie jednocześnie) w trakcie swojego cyklu życia;
  - *zachowanie* - zbiór zaimplementowanych metod, które są przypisane do obiektu.

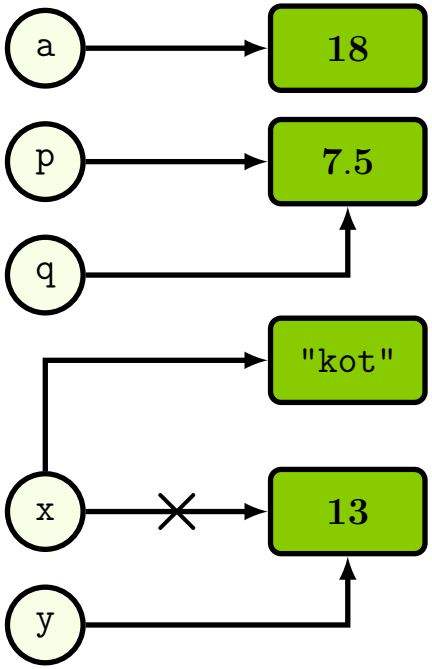
    PRZYKŁAD 1
- Optymalizacja dla liczb całkowitych i ciągów tekstowych w konsoli interaktywnej (Jupyter Notebook).

Zmienne w Pythonie

- **Zmienne** w Pythonie posiadają nazwę, wartość oraz zasięg. Zmienne nie posiadają typu, gdyż jest to cecha obiektu z którym aktualnie zmienna jest związana. Zmienne w Pythonie:
  - zawsze muszą mieć jakąś wartość (chociażby `None`) - brak niezainicjalizowanych zmiennych;
  - same z siebie nie powodują alokacji pamięci;
  - tak naprawdę zmienne to pewne odniesienia czyli referencje (etykiety) do obiektów.

```
>>> a = 18
>>> p = 7.5
>>> q = p


>>> x = 13
>>> y = x
>>> x = "kot"
```



Zmienne w Pythonie

- Type niezmienne

  PRZYKŁAD 2

- Type mutowalne

  PRZYKŁAD 3

- Kopiowanie płytkie i głębokie

  PRZYKŁAD 4

  PRZYKŁAD 5

- Mechanizmy automatycznego zarządzania pamięcią:

  - zliczanie referencji;
  - odśmiecanie pamięci (ang. *garbage collection*).

    PRZYKŁAD 6

    PRZYKŁAD 7

Typy w Pythonie

- Python jest językiem należącym do klasy języków dynamicznych. Należy pamiętać, że nie oznacza to Pythone w ogóle nie stosuje typów lub obiekty mogą zmieniać typ dynamicznie. Raz utworzony obiekt Pythone nie może zmienić typu ani tożsamości. Dynamizm Pythona oznacza, że kontrola typów odbywa się w czasie wykonywania programów, a nie w czasie kompilacji (C, C++, Java). Python jest językiem interpretowanym. Wadą takiego podejścia jest późne wykrywanie błędów związanych z niezgodnością typów (czasami błędy pozostają "uśpione"). Zalety dynamicznego typowania to m.in.:
  - mniejsza ilość kodu;
  - małpie łatanie - dynamiczne zmiana kodu, tworzenie atrap obiektów (ułatwia testowanie);
  - metaprogramowanie - dynamiczna analiza, wczytywanie, modyfikowanie i ewaluacja kodu (`eval`);

Typy w Pythonie

- Python jest językiem w którym występuje silan kontrola typów (podobnie jak Java). Językiem stycznie typowanym z słabą kontrolą typów jest np język C. W języku silnie typowanym nie występują automatyczne i niejawne konwersje między typami niespokrewnionymi:

```
>>> 3 + '4'
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    3 + '4'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> str(3) + '4', 3 + int('4')
('34', 7)
```

- Python stosuje trochę rozluzioną kontrolę typów:

```
>>> 1 == True, 1 == 1.0, {True: "a", 1: "b", 1.0: "c"}
(True, True, {True: 'c'})
```

- Python wykorzystuje tzw. **kacze typowanie** (ang. *duck typing*) - wyłącznie kontroli typów w czasie przekazywania argumentów do funkcji. **PRZYKŁAD 8**

Programowanie obiektowe

- **Programowanie obiektowe** (ang. *object-oriented programming* - ***OOP***)
  - podstawowy element konstrukcyjny w programowaniu OOP to obiekt programowy – zwany po prostu obiektem.
  - atuty programowania, projektowania oraz analizy obiektowej?
  - obiekty programowe łączą w sobie:
    - cechy (zwane w terminologii OOP **atrybutami**)
    - zachowania (zwane w języku OOP **metodami**)
- Programowanie obiektowe polega m.in. na definiowaniu abstrakcji przy pomocy własnych typów danych, które modelują pewien fragment rzeczywistości. Python pozwala definiować nowe typy danych na dwa sposoby: statycznie (klasy) i dynamicznie (funkcja `type`).
- Statyczne tworzenie obiektów
  - **konkretyzowanie** obiektów na podstawie ich definicji - **klasy** (plan/wzorzec obiektu)
  - obiekt, czyli egzemplarz lub instancja klasy

Definiowanie klas w języku Python

- Tworzenie własnych klas
  - słowo kluczowe `class`
  - konwencja nazewnicza klas
  - składnia Pythona służąca do definiowania własnych klas:

```
class NazwaKlasy1:
    pakiet


class NazwaKlasy2(klasa_bazowa):
    pakiet
```

  - dokumentujący ciąg tekstowy

Konkretyzacja obiektu

- Tworzenie nowych obiektów na podstawie danej klasy **PRZYKŁAD 9**
- Definiowanie metod
  - parametr `self` dla **metody instancji**
  - wywoływanie własnych metod, metody specjalne
  - metoda specjalne `__init__()` oraz `__new__()`
  - własny **konstruktor** - metoda specjalna `__init__()`
  - atrybuty i dostęp do nich - wykorzystanie konstruktora

  PRZYKŁAD 10

  PRZYKŁAD 11

  PRZYKŁAD 12
- Wyświetlanie obiektu
  - metoda specjalne `__str__()`

    PRZYKŁAD 13

Atrybuty i metody klasy

- Wartości związane z samą klasą tzw. **atrybuty klasy**
  - tworzenie atrybutów klasy
- Metody związane z daną klasą tzw. **metody statyczne**
  - tworzenie metod klasy
  - dekorator `@staticmethod`
  - dekoratory **przykład dekoratora**

PRZYKŁAD 14

Atrybuty i metody chronione oraz prywatne

- Wspieranie hermetyzacji obiektów - w Pythonie wszystko jest zawsze dostępne publiczne (brak modyfikatorów widoczności typu *private* lub *protected*), w pythonie obowiazują pewne niepisane konwencje odnośnie prefiksowania nazw.
  - atrybuty chronione - dostęp możliwy tylko z poziomu tej samej klasy oraz klas pochodnych (to jest tylko umowa)
  - atrybuty prywatne - dostęp możliwy tylko z poziomu metod tej samej klasy (dostęp i tak jest możliwy)
  - metody prywatne i chronione
  - używanie atrybutów oraz metod chronionych i prywatnych a filozofia Pythona

  PRZYKŁAD 15

Odpowiednikiem atrybutów i metod prywatnych np. języka C++ są w języku Python atrybuty i metody chronione (ich nazwy poprzedzamy jednym znakiem podkreślenia). Pythonowe atrybuty i metody prywatne należy stosować tylko wtedy gdy chcemy się zabezpieczyć przed ich przesłonieniem w wyniku dziedziczenia.

Kontrolowanie dostępu do atrybutów

- Właściwości - dekorator `@property`
  - tworzenie atrybutów tylko do odczytu

    PRZYKŁAD 16
  - dynamiczne obliczanie wartości atrybutów - funkcjonalność metody a wygląd atrybutu

    PRZYKŁAD 17
  - właściwości a kontrola poprawność wartości atrybutów

    PRZYKŁAD 18

    PRZYKŁAD 19

Przechowywanie złożonych struktur danych w plikach – moduł `pickle`

- Marynowanie w Pythonie, czyli tzw. **serializacjia**
  - marynowanie - funkcja `pickle.dump()`
  - odmarynowaanie - funkcja `pickle.load()`

PRZYKŁAD 20

PRZYKŁAD 21

Przechowywanie złożonych struktur danych w plikach – moduł `shelve`

- **Półka** (ang. *shelf*), czyli coś jak słownik na obiekty
  - tworzenie półki - funkcja `shelve.open()`
  - klucze półki - tylko ciągi tekstowe!
  - tryb dostępu do półki:

Tryb	Opis
`"c"`	Otwiera plik do odczytu i zapisu. Jeżeli plik nie istnieje, zostaje utworzony. **Uwaga:** Jest to domyślny tryb otwierania pliku funkcji `shelve.open()`.
`"n"`	Tworzy nowy plik do odczytu i zapisu. Jeżeli plik już istnieje, jego zwartość zostaje nadpisana.
`"r"`	Odczyt danych z pliku. Jeżeli plik nie istnieje, Python zgłosi błąd (wyjątek).
`"w"`	Zapis danych do pliku. Jeżeli plik nie istnieje, Python zgłosi błąd (wyjątek).

PRZYKŁAD 22