Wykłady ze Wstępu do Informatyki II

dla kierunku Matematyka

czyli nauka programowania w języku Python

dr Adam Marszałek

Instytut Informatyki
Politechnika Krakowska

Rok akademicki 2017/2018

Liczy pseudolosowe - moduł `random`

Generatory liczb pseudolosowych:

- Nie jest możliwe wygenerowanie liczby prawdziwie losowej.
- Możliwe jest wygenerowanie (na bazie liczby wejściowej, **seed**) ciągu liczb pseudolosowych.
  - Ten sam seed = ten sam ciąg.
  - Maksymalna ilość ciągów = możliwe seedy.
  - Okresowość.

Liczby pseudolosowe w Pythonie - moduł `random`

- Moduł implementuje generatory liczb pseudolosowych o różnych rozkładach.
- Umożliwia wybór losowej liczby całkowitej z zadanego zakresu jak również losowego elementu z podanej sekwencji przy założeniu równomiernego rozkładu prawdopodobieństwa.
- Implementuje również funkcję generującą losową permutację listy, działającą w miejscu, oraz funkcję wybierającą losowe elementy listy bez zwracania.

Liczy pseudolosowe - moduł `random`

- Oferuje również funkcje do obliczania rozkładów: jednorodnego, normalnego (Gaussa), długiego normalnego, ujemnego wykładniczego, gamma i beta.
- Prawie wszystkie funkcje modułu korzystają z bazowej funkcji `random()`, która generuje zmiennoprzecinkowe liczby pseudolosowe z lewostronnie domkniętego przedziału $[0.0, 1.0)$.
- Język Python używa generatora o nazwie **Mersenne Twister**, dzięki któremu możliwe jest uzyskanie liczb zmiennoprzecinkowych o 53-bitowej precyzji, przy czym okres tego generatora wynosi $2^{19937} - 1$.
- Implementacja w języku C, na której generator ów bazuje, jest szybka i bezpieczna ze względu na wątki.
- Mersenne Twister jest jednym z najszerzej badanych generatorów spośród wszystkich obecnie istniejących. Jednakże, ze względu na fakt, że jest on **całkowicie deterministyczny**, nie nadaje się do wszystkich celów, a w szczególności nie nadaje się do celów kryptograficznych.

Moduł `random` - Funkcje dla liczb całkowitych

- `randrange([start,] stop[, krok])` - Zwraca element losowo wybrany z zakresu `range(start, stop, krok)`. Znaczenie argumentów i ich wymagalność jest taka sam jak w metodzie `range()` tzn. można wywołać ją z jednym argumentem: `stop`, dwoma: `start`, `stop` oraz trzema: `stary`, `stop`, `krok`.
- `randint(a, b)` - Zwraca losową liczbę całkowitą N spełniającą nierówność $a \leqslant N \leqslant b$.

In [1]:
```
import random
losowa = random.randint(3, 10)
losowe1 = [random.randint(3, 10) for _ in range(10)]
losowe2 = [random.randrange(3, 11, 3) for _ in range(10)]
print(losowa, losowe1)
print(losowe2)
```

```
7 [5, 6, 8, 4, 3, 6, 6, 10, 6, 7]
[9, 6, 9, 6, 6, 9, 6, 3, 9, 3]
```

Moduł `random` - Funkcje dla sekwencji

- `choice(seq)` - Zwraca losowo wybrany element niepustej sekwencji `seq`.
- `shuffle(x[, random])` - Miesza elementy sekwencji `x` w miejscu. Opcjonalny argument `random` jest zeroargumentową funkcją zwracającą losową liczbę zmiennoprzecinkową z przedziału $[0.0, 1.0)$. Domyślnie używana jest funkcja `random()`. Należy zauważyć, ze nawet dla małych wartości `len(x)`, całkowita liczba permutacji sekwencji `x` jest większa, niż cykl większości generatorów liczb losowych. Oznacza to, że w przypadku długich sekwencji większość permutacji nie zostanie wygenerowana.
- `sample(populacja, k)` - Zwraca listę o długości `k` zawierającą unikatowe elementy wybrane z sekwencji reprezentującej populację. Używana do wybierania losowych elementów bez zwracania. Zwrócona lista zawiera elementy wybrane z przekazanej jako argument populacji, jednak oryginalna populacja pozostaje niezmieniona. Lista wynikowa zawiera elementy uporządkowane w kolejności wynikającej z kolejności losowania elementów z populacji.

Moduł `random` - Funkcje dla sekwencji

In [1]:
```
import random

x = "Język Python"
y = ['a', 'b', 'c', 1, 2, 3]
```

In [2]:
```
random.choice(x)
```

Out[2]:
```
'z'
```

In [3]:
```
random.sample(y, 3)
```

Out[3]:
```
[3, 'c', 1]
```

In [4]:
```
print(y)
random.shuffle(y)
print(y)
```

```
['a', 'b', 'c', 1, 2, 3]
[1, 'c', 'b', 'a', 3, 2]
```

In [5]:
```
print(x)
z=random.sample(x, len(x))
print(''.join(z))
```

```
Język Python
tnhkzyęo yJP
```

Moduł `random` - Funkcje dla liczb rzeczywistych

- `random()` - Zwraca kolejną losową liczbę zmiennoprzecinkową z zakresu $[0.0, 1.0)$.
- `uniform(a, b)` - Zwraca losową liczbę rzeczywista N spełniającą nierówność $a \leqslant N < b$.
- `gauss(mu, sigma)` - Rozkład Gaussa. Argument `mu` jest średnią rozkładu, a argument `sigma` określa odchylenie standardowe. Funkcja ta jest nieznacznie szybsza od zdefiniowanej poniżej funkcji `normalvariate()`.
- `normalvariate(mu, sigma)` - Rozkład normalny. Argument `mu` jest średnią, a argument `sigma` jest odchyleniem standardowym.

Pozostałe metody. Nazwy parametrów funkcji pochodzą od nazw odpowiednich zmiennych w równaniach opisujących rozkład:

- `triangular(low, high, mode)`, `betavariate(alpha, beta)`, `expovariate(lambd)`,`gammavariate(alpha, beta)`, `lognormvariate(mu, sigma)`, `vonmisesvariate(mu, kappa)`, `paretovariate(alpha)`, `weibullvariate(alpha, beta)`.

Moduł `random` - Funkcje dla liczb rzeczywistych

In [1]:
```
import random
losowe = [random.random() for _ in range(5)]
print(losowe)
```

```
[0.12346497985828264, 0.02565278392819803, 0.8064623931033604, 0
.45962501736059913, 0.24070115550768023]
```

In [2]:
```
losowe = [random.uniform(99, 100) for _ in range(5)]
print(losowe)
```

```
[99.22622919829044, 99.07930538336072, 99.22215744854459, 99.941
48516795431, 99.05330398627291]
```

In [3]:
```
losowe = [random.gauss(3, 1) for _ in range(5)]
print(losowe)
```

```
[4.110704531451148, 3.366802914506136, 1.8426468910545357, 3.912
043633054523, 3.077878802998786]
```

In [4]:
```
from statistics import mean, stdev
losowe = [random.gauss(3, 1) for _ in range(1000)]
print(mean(losowe), stdev(losowe))
```

```
3.0439901161273886 1.0019937849679506
```

Moduł `random` - Funkcje rejestrujące stan

- `seed(a=None, version=2)` - Inicjalizuje podstawowy generator liczb losowych. Jeśli `x` zostanie pominięty lub ma wartość `None`, zostanie użyty bieżący czas; bieżący czas używany jest również wówczas, gdy moduł jest importowany po raz pierwszy. Jeżeli argument ma wartość całkowitoliczbową, wartość ta używana jest do zainicjalizowania w sposób bezpośredni. Argument `x` może być dowolnym obiektem (patrz dokumentacja).
- `getstate()` - Zwraca obiekt reprezentujący bieżący wewnętrzny stan generatora. Obiekt ten może być przekazany metodzie `setstate()` w celu przywrócenia pobranego wcześniej stanu.
- `setstate(stan)` - Argument `stan` powinien pochodzić z wcześniejszego wywołania `getstate()`. Funkcja `setstate()` przywraca generatorowi jego wewnętrzny stan, w jakim generator znajdował się w momencie wywołania metody `getstate()`.

Moduł `random` - Funkcje rejestrujące stan

In [1]:
```
import random
for _ in range(4):
    print(random.random())
```

```
0.6345011905140988
0.06753727236581897
0.713921617453268
0.7152108465098379
```

In [2]:
```
for _ in range(4):
    random.seed(1234) # stały seed -> stała wartość
    print(random.random())
```

```
0.9664535356921388
0.9664535356921388
0.9664535356921388
0.9664535356921388
```

In [3]:
```
random.seed()
state = random.getstate()
for i in range(6):
    if i==3:
        random.setstate(state)
    print(random.random())
```

```
0.14984893890992002
0.4483378274372779
0.21312279735627115
0.14984893890992002
0.4483378274372779
0.21312279735627115
```

Programowanie obiektowe - ogólnie

Programowanie obiektowe

- **Object-oriented programming**
- Najpopularniejszy obecnie styl (paradygmat) programowania.
- Rozwinięcie koncepcji programowania strukturalnego.
  - Podejście tradycyjne: program jako kolekcja funkcji (a wcześniej lista instrukcji).
  - Podejście obiektowe: program jako kolekcja współpracujących ze sobą obiektów.
- Obiekty łączą dane i operacje na nich.
- Zalety programowania obiektowego:
  - Ułatwia współpracę i podział zadań między programistów.
  - Ułatwia pielęgnację i rozbudowę aplikacji.
  - Ułatwia ponowne wykorzystywanie wcześniej napisanego kodu.
  - Często umożliwia naturalne modelowanie rzeczywistości.
  - Odpowiednie dla dużych projektów, popularne w inżynierii oprogramowania.

Historia programowania obiektowego

- **Simula** (Simula I, Simula 67)
  - Pierwszy zorientowany obiektowo język programowania.
  - Opracowany w Norwegian Computer Center na bazie języka Algol 60.
  - Język opracowany z myślą o symulacjach (symulacje statków).
  - Wprowadził pojęcia klas, obiektów, podklas.
- **Smalltalk** (Smalltalk 80)
  - Opracowany w Xerox Palo Alto Research Center.
  - Graficzny interfejs użytkownika.
  - Język w pełni obiektowy („wszystko jest obiektem").
  - Mechanizm reflekcji (programowy odczyt struktury klasy).
  - Wykorzystywany głównie do celów dydaktycznych.
- Następnie cała masa języków hybrydowych: C++, Java, C#, Python, Ruby itp.

Obiekty i Klasy

- **Klasa** to złożony typ danych składający się z pól przechowujących dane oraz posiadający metody wykonujące zaprogramowane czynności.
- **Obiekt** może reprezentować cokolwiek. Każdy obiekt należy do pewnej klasy. Definicja klasy zawiera pola z których składa się ów obiekt, oraz metody, którymi dysponuje.
- Klasa jest więc wzorcem na podstawie którego powołujemy do życia obiekty (instancje klasy).
- Obiekt znajduje się w pamięci w trakcie wykonywania programu, a klasa jest wzorcem/szablonem.

Pola i Metody

- Obiekty zawierają **pola**, czyli zmienne. Ich rolą jest przechowywanie pewnych informacji o obiekcie – jego charakterystyki.
- Obiekt może wykonywać na sobie pewne działania, a więc uruchamiać zaprogramowane funkcje. Nazywamy je **metodami** albo funkcjami składowymi. Czynią one obiekt tworem, aktywnym – nie jest on jedynie pojemnikiem na dane, lecz może samodzielnie nimi manipulować.
- Metoda w programowaniu obiektowym jest to funkcja składowa klasy, której zadaniem jest działanie na rzecz określonych elementów danej klasy lub klas z nią spokrewnionych.
- Metody wiąże się z klasami głównie po to aby nie zaśmiecać kodu źródłowego i samego programu nadmierna ilością funkcji globalnych, które i tak nie zostaną użyte w innym celu niż na rzecz konkretnej klasy.
- Inną zaletą metod jest to, że metoda wewnętrzna danej klasy ma dostęp do wszystkich składników tej klasy (także prywatnych i chronionych) bez konieczności deklarowania np. zaprzyjaźnienia.

Paradygmaty programowania obiektowego - Abstrakcja

Abstrakcja:

- Bierzemy świat rzeczywisty grupujemy go w elementy, elementy stają się obiektami, obiektom przypisujemy zachowanie, stan.
- Umiejętność wyodrębniania cech istotnych dla danego problemu.
- Praca z obiektami na poziomie ogólności (względem hierarchii dziedziczenia) odpowiednim dla rozwiązywanego problemu.

Klasa abstrakcyjna:

- Nie posiada obiektów, czyli nie posiada stanu, posiada metody i interfejsy.
- Używana do definicji interfejsu dla klas z niej wywiedzionych.

Metoda abstrakcyjna:

- Nie została jeszcze zaimplementowana (tylko zadeklarowana, aby zaimplementować ją w klasach potomnych).
- Deklarowana jedynie wewnątrz klasy abstrakcyjnej.

Paradygmaty programowania obiektowego - Enkapsulacja

Enkapsulacja (hermetyzacja):

- Ukrycie szczegółów implementacji klasy przed kodem korzystającym z klasy. Zapewnia, że obiekt nie może zmienić stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko wewnętrzne metody obiektu są uprawnione do zmiany jego stanu. Każdy typ obiektu prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.
- Pewne języki osłabiają to założenie, dopuszczając pewien poziom bezpośredniego (kontrolowanego) dostępu do „wnętrzności" obiektu. Ograniczają w ten sposób poziom abstrakcji.

Poziomy kontroli dostępu:

- public – dostępne dla każdego (to co jest w kodzie klasy będzie widoczne na zewnątrz).
- private – tylko twórca klasy.
- protected – dostęp mają tylko dana klasa i klasy dziedziczące.

Paradygmaty programowania obiektowego - Dziedziczenie

Dziedziczenie:

- Definiowanie podklasy (klasy pochodnej) jako specjalizacji klasy już istniejącej (klasy bazowej, nadklasy).
- Podklasa dziedziczy z klasy bazowej atrybuty i metody.
- Cechy wspólne dla wszystkich podklas definiowane są w nadklasie.
- Podklasa może posiadać dodatkowe atrybuty i metody oraz redefiniować metody odziedziczone.

Dziedziczenie wielobazowe (wielorakie):

- Polega na definiowaniu klasy jako dziedziczącej bezpośrednio z więcej niż jednej klasy.
- Dostępne nie we wszystkich językach.
- Prowadzi do skomplikowanych programów.

Paradygmaty programowania obiektowego - Polimorfizm

Polimorfizm:

- Technika ściśle związana z dziedziczeniem – zwykle uważana za najważniejszą zaletę technologii obiektowych.
- Różne zachowanie w odpowiedzi na takie samo wywołanie metody w zależności od konkretnej klasy obiektu.
- Od strony technicznej sprowadza się do tzw. **późniejszego wiązania** - decyzja o tym, z której klasy metodę wywołać podejmowana jest w trakcie działania programu, a nie na etapie kompilacji.

Unified Modeling Language

Unified Modeling Language (UML):

- Język do modelowania obiektowego.
- Język ogólnego przeznaczenia, niezależny od języków programowania obiektowego.
- Opracowany przez Object Management Group (OMG).
- Wykorzystywany w inżynierii oprogramowania.
- Standaryzuje graficzną notację do tworzenia abstrakcyjnego modelu systemu.
- Modele UML mogą być automatycznie transformowane do kodu w konkretnym języku programowania np. Java, Python.

Składniki modelu systemu w UML

Model funkcjonalny:

- Reprezentuje funkcjonalność systemu z punktu widzenia użytkownika.
- Zawiera m.in. diagramy przypadków użycia.

Model obiektowy:

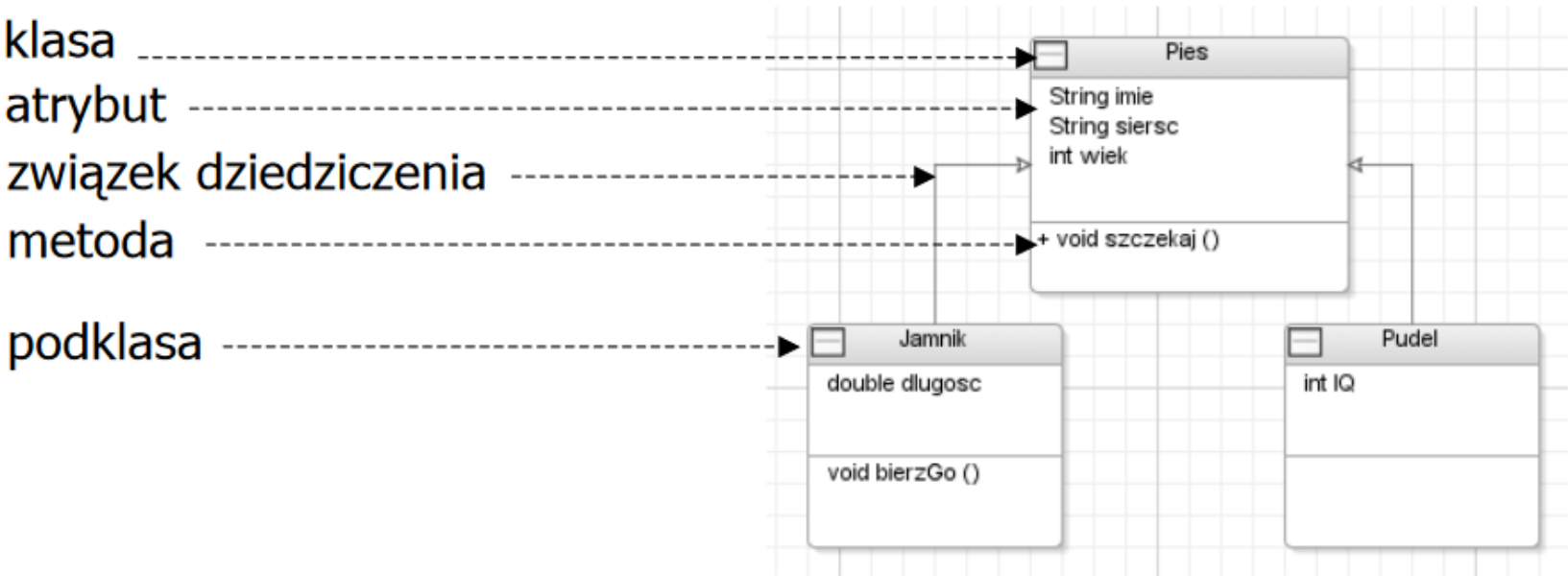
- Reprezentuje strukturę systemu, obejmującą obiekty, atrybuty, operacje i powiązania.
- Zawiera m.in. diagramy klas.

Model dynamiczny:

- Reprezentuje wewnętrzne działanie systemu.
- Zawiera m.in. diagramy aktywności (przepływu sterowania).

Diagramy UML

- Diagram stanowi częściową, graficzną reprezentację modelu (najczęściej uzupełnia go dokumentacja tekstowa)
- Przykładowy diagram klas:



Programowanie obiektowe w języku Python

Składnia definicji klasy

- Do definicji klasy służy słowo kluczowe `class`.

```
In [1]: class NazwaKlasy:
            pass
```

- W Pythonie każda klasa dziedziczy po klasie `object`. W Pythonie 3 można ale nie trzeba tego podawać tzn. można zdefiniować klasę w taki sposób:

```
In [2]: class NazwaKlasy(object):
            pass
```

- Równoważnie do powyższych można napisać też coś takiego:

```
In [3]: class NazwaKlasy():
            pass
```

- W Pythonie 2 podane wyżej konstrukcje nie są równoważne (patrz zagadnienie: „old vs. new-style python objects").

Metody i atrybuty (pola) klasy

- **Metoda klasy** to funkcja zdefiniowana wewnątrz klasy.
- **Atrybut klasy** to zmienna zdefiniowana wewnątrz klasy.

```
In [1]: class PierwszaKlasa:
            imie = 'Adam'  # atrybut klasy
            def przywitanie(self):  # metoda klasy
                print('Witaj {}'.format(self.imie))
```

- Pierwszy argument każdej metody – `self` – jest referencją do obiektu, na rzecz którego ta metoda została wywołana.
- W Pythonie przyjęto konwencję `self` ale może to być dowolna nazwa, np. `this` jak w C++.

Odniesienie do atrybutów klasy

- Kiedy wprowadza się definicję klasy do programu, tworzona jest nowa przestrzeń nazw używana jako zasięg lokalny nazw, w ten sposób, wszystkie przypisania do zmiennych lokalnych dotyczą nazw z tej właśnie przestrzeni.
- Kiedy definicja klasy się kończy, tworzony jest obiekt klasy (nie jest to instancja klasy).
- **Odniesienie do atrybutu klasy** odbywa się poprzez standardową konstrukcję `obiekt.nazwa`. Prawidłowymi nazwami atrybutów są nazwy, które istniały w przestrzeni nazw klasy w czasie tworzenia jej obiektu.

```
In [2]: PierwszaKlasa.imie
Out[2]: 'Adam'

In [3]: PierwszaKlasa.przywitanie
Out[3]: <function __main__.PierwszaKlasa.przywitanie>
```

Konkretyzacja klasy

- **Konkretyzację klasy** przeprowadza się używając notacji wywołania funkcji. Należy tylko udać, że obiekt klasy jest bezparametrową funkcją, która zwraca **instancję (konkret)** klasy.

```
In [4]: a = PierwszaKlasa()

In [5]: a.imie
Out[5]: 'Adam'

In [6]: a.przywitanie()
        Witaj Adam

In [7]: PierwszaKlasa.przywitanie(a)
        Witaj Adam
```

- Podczas wywołania metody obiektu nie wymieniamy na liście argumentów argumentu `self`. Jest on automatycznie przekazywany przez Pythona do metody.

Inicjalizacja

- Operacja konkretyzacji tworzy pusty obiekt. Dla wielu klas występuje konieczność stworzenia swojego konkretu w pewnym znanym, początkowym stanie. Służy do tego wbudowana metoda `__init__()`.
- W momencie konkretyzacji klasy, automatycznie wywołana zostanie metoda `__init__()` dla nowopowstałego konkretu klasy.
- Prawie jak konstruktor, ale wywoływana **po** utworzeniu instancji.
- Zmienne instancji definiujemy w metodzie `__init__()`.
- W Pythonie można dodawać nowe atrybuty do istniejących obiektów (choć nie jest to zalecane!).
- Nazwy i wartości zdefiniowanych zmiennych obiektu znajdują się w atrybucie wbudowanym `__dict__`.

Inicjalizacja

In [1]:
```
class PierwszaKlasa:
    def __init__(self, imie):
        self.imie = imie  # atrybut klasy

    def przywitanie(self):  # metoda klasy
        print('Witaj {}'.format(self.imie))

a = PierwszaKlasa('Adam')
b = PierwszaKlasa('Basia')
print(a.imie, b.imie)
a.przywitanie()
b.przywitanie()
```

```
Adam Basia
Witaj Adam
Witaj Basia
```

In [2]:
```
a.nazwisko = 'Marszałek'
a.__dict__
```

Out[2]:
```
{'imie': 'Adam', 'nazwisko': 'Marszałek'}
```

In [3]:
```
b.__dict__
```

Out[3]:
```
{'imie': 'Basia'}
```

Wbudowane metody i atrybuty specjalne

- Nazwy metod specjalnych zawsze rozpoczynają się i kończą dwoma znakami podkreślenia. Python uruchamia te metody automatycznie, gdy interpreter napotka funkcję wbudowaną lub operator odpowiadający danej metodzie specjalnej. Python używa metod specjalnych w następujących przypadkach:
  - tworzenie i usuwanie egzemplarza,
  - tworzenie reprezentacji łańcuchowych egzemplarza,
  - definiowanie wartości prawdziwości egzemplarza;
  - porównywanie egzemplarzy;
  - dostęp do atrybutów egzemplarza;
  - traktowanie egzemplarzy jak sekwencji i słowników;
  - wykonywanie operacji matematycznych na egzemplarzach.
- **Uwaga:** Nie można używać metod specjalnych do zmiany zachowania wbudowanych typów.

Dokumentacja klasy

- Na początku definicji klasy może my umieścić wiersz dokumentacyjny.
- Łańcuch dokumentacyjny jest przypisywany jako wartość atrybutu __doc__. Jeżeli nie zostanie podany łańcuch dokumentacyjny to wartością atrybutu __doc__ jest None.
- Łańcuchy dokumentacyjne można dodawać do metod klasy w taki sam sposób jak do funkcji.

In [1]:
```
class PierwszaKlasa:
    """Nasza pierwsza klasa w języku Python"""
    def __init__(self, imie):
        self.imie = imie
    def przywitanie(self):
        """Metoda w klasie PierwszaKlasa"""
        print('Witaj {}'.format(self.imie))

a = PierwszaKlasa('Adam')
a.__doc__
```

Out[1]:
```
'Nasza pierwsza klasa w języku Python'
```

In [2]:
```
a.przywitanie.__doc__
```

Out[2]:
```
'Metoda w klasie PierwszaKlasa'
```

Dokumentacja klasy

- Do wyświetlenia informacji o obiekcie klasy można użyć też metody wbudowanej `help()`.

```
In [3]: help(a)
Help on PierwszaKlasa in module __main__ object:

class PierwszaKlasa(builtins.object)
 |  Nasza pierwsza klasa w języku Python
 |
 |  Methods defined here:
 |
 |  __init__(self, imie)
 |      Initialize self.  See help(type(self)) for accurate sig
nature.
 |
 |  przywitanie(self)
 |      Metoda w klasie PierwszaKlasa
 |
 |  ----------------------------------------------------------------
-----------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Przeciążanie funkcji

- Przeciążanie funkcji polega na tym, że funkcja o takiej samej nazwie zachowuje się różnie w zależności od typów lub klas podanych argumentów. Python wybiera odpowiednie działanie automatycznie na podstawie parametrów funkcji, co znaczy, że na przykład wywołuje metodę specjalną `__str__(x)` po wywołaniu `str(x)`, jeśli `x` jest egzemplarzem klasy dysponującej metodą `__str__()`.
- Przeciążyć możemy np. metody:
- `__repr__()` - „oficjalna" reprezentacja obiektu, powinna być jednoznaczna; wywołana przez `repr(obiekt)` lub w interpreterze po wpisaniu nazwy zmiennej.
- `__str__()` - „nieformalna" reprezentacja obiektu, powinna być czytelna; wywołana przez `str(obiekt)` lub `print`.

Metody `str` i `repr`

- Klasa bez przeciążonych metod `__repr__()` i `__str__()`.

In [1]:
```
class PierwszaKlasa:
    def __init__(self, imie):
        self.imie = imie
    def przywitanie(self):
        print('Witaj {}'.format(self.imie))

a = PierwszaKlasa('Adam')
a  # wykonywana jest metoda repr
```

Out[1]:
```
<__main__.PierwszaKlasa at 0xd5106b390>
```

In [2]:
```
# wewnątrz metody print wykonywana jest metoda str
print(a)
```

```
<__main__.PierwszaKlasa object at 0x0000000D5106B390>
```

Metody str i repr

- Klasa z przeciążonymi metodami __repr__() i __str__().

In [1]:
```
class PierwszaKlasa:
    def __init__(self, imie):
        self.imie = imie
    def przywitanie(self):
        print('Witaj {}'.format(self.imie))
    def __str__(self):
        return 'Obiekt PierwszaKlasa, imie = {}'.format(self.imie)
    def __repr__(self):
        return 'PierwszaKlasa({})'.format(self.imie)

a = PierwszaKlasa('Adam')
a  # wykonywana jest metoda repr
```

Out[1]:
```
PierwszaKlasa(Adam)
```

In [2]:
```
# wewnątrz metody print wykonywana jest metoda str
print(a)
```

```
Obiekt PierwszaKlasa, imie = Adam
```

Atrybuty i metody prywatne

- Przyjmując terminologię z innych języków obiektowych np. C++, wszystkie składowe klasy w Pythonie **są publiczne**.
- Python posiada jednak pewien ograniczony mechanizm implementacji zmiennych prywatnych klasy.
- Zmienne prywatne sygnalizuje się poprzez poprzedzenie nazwy znakiem podkreślenia _.

In [1]:
```
class PierwszaKlasa:
    def __init__(self, imie):
        self._imie = imie  # atrybut prywatny
    def _przywitanie(self):  # metoda prywatna
        print('Witaj {}'.format(self._imie))

a = PierwszaKlasa('Adam')
```

In [2]:
```
a._przywitanie()
```

```
Witaj Adam
```

In [3]:
```
a._imie = 'Basia'
a._imie
```

Out[3]:
```
'Basia'
```

Atrybuty i metody bardziej prywatne

- Jak widać dostęp do zmiennych „niby” prywatnych nie jest ograniczony, jest to tylko sygnalizacja, żeby te zmienne traktować jak prywatne.
- W Pythonie możemy zdefiniować również tzw. zmienne bardziej prywatne poprzez poprzedzenie nazwy dwoma znakami podkreślenia __.

In [1]:
```
class PierwszaKlasa:
    def __init__(self, imie):
        self.__imie = imie  # atrybut prywatny
    def __przywitanie(self):  # metoda prywatna
        print('Witaj {}'.format(self.__imie))

a = PierwszaKlasa('Adam')
```

Atrybuty i metody bardziej prywatne

```
In [2]: a.__przywitanie()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-453544c82b24> in <module>()
----> 1 a.__przywitanie()

AttributeError: 'PierwszaKlasa' object has no attribute '__przywitanie'
```

```
In [3]: a.__imie
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-ab0638310088> in <module>()
----> 1 a.__imie

AttributeError: 'PierwszaKlasa' object has no attribute '__imie'
```

Atrybuty i metody bardziej prywatne

- Jak widać Python zgłasza wyjątek `AttributeError` przy próbie dostępu z zewnątrz klasy do nazwy prywatnej.
- Mogło by się więc wydawać że mamy ograniczony dostęp do zmiennych bardziej prywatnych jednak w rzeczywistości tak nie jest.
- Python ukrywa nazwę prywatną zmieniając jej nazwę wewnętrzną na `_classname__attrname`.
- Taki mechanizm zabezpieczania nazywany przekręcaniem nazw jest w istocie iluzją, bowiem można uzyskać dostęp do prywatnego atrybutu posługując się jego przekręconą nazwą.

```
In [4]: a._PierwszaKlasa__imie
Out[4]: 'Adam'

In [5]: a._PierwszaKlasa__przywitanie()
        Witaj Adam
```