Faculty of Geodesy and Cartography

WARSAW UNIVERSITY OF TECHNOLOGY

Informatyka Geodezyjna - wykłady/ćwiczenia, rok akad. 2021-2022

Wyk. 13-14: Python z klasą - wprowadzenie do programowania obiektowego

Kinga Węzka
kinga.wezka@pw.edu.pl
Zakład Geodezji i Astronomii Geodezyjnej

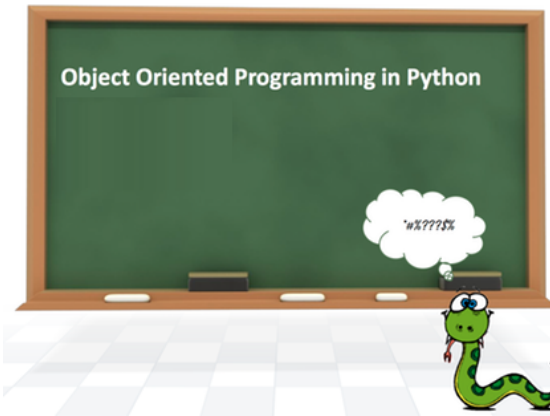
Warsaw University of Technology

Plan

1. Programowanie obiektowe – wprowadzenie
2. Notacja obiektowa: Klasa, obiekt, konkretyzacja, instancja
3. Metody specjalne
4. Podstawowe założenia paradygmatu obiektowego
   - Dziedziczenie
     - Dziedziczenie – funkcja super
     - Kontrola dziedziczenia
   - Polimorfizm
   - Abstrakcja
   - Hermetyzacja
5. Modyfikatory dostępu: public, private, protected
   - Modyfikatory dostępu: public (publiczny)
   - Modyfikatory dostepu: protected (chroniony)
   - Modyfikatory dostępu: private (prywatny)
6. Podsumowanie – terminologia OPP
   - Literatuta, tutoriale, pomoce

Programowanie obiektowe – wprowadzenie

- Programowanie obiektowe, czy szerzej programowanie zorientowane obiektowo (ang. *object-oriented programming*, OPP), jest pewnym sposobem organizacji kodu
- Jest to paradygmat programowania, w którym programy definiuje się za pomocą **klas, obiektów** – elementów łączących stan (czyli dane, nazywane najczęściej **polami lub atrybutami**) i zachowanie (czyli funkcje, tu: **metody**).
- Programowanie obiektowe ma ułatwić pisanie, konserwację, aktualizację i wielokrotne użycie programów lub ich fragmentów.




I have class

- Python jest języykiem obiektowym od początku swojego istnienia
- W Pythonie utworzenie zmiennej jest równoznaczne i przypisanie jej wartości (i = 5) oznacza utworzenie **obiektu (zmienna jest obiektem)** o nazwie (i), będącego obiektem (intstancją) **klasy** int.

In[1]:

```
1 i = 5
```

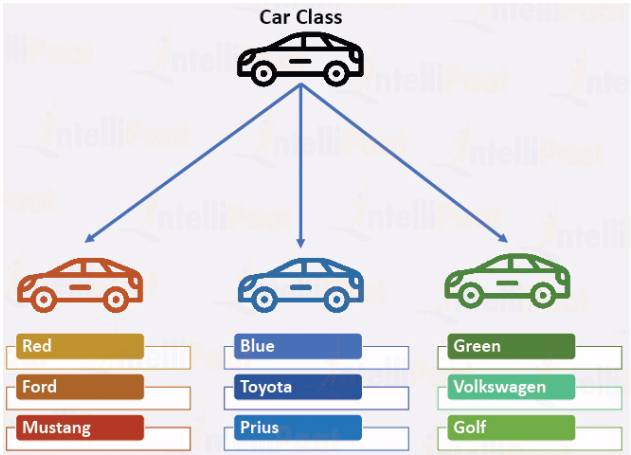
- Natomiast **atrybuty/pola** obiektu klasy **int**.

In[2]:

```
1 >>> a = 1
2 >>> print('a = ', a.numerator, '/', a.denominator)
3 a =  1 / 1
4 >>> print('a = ', a.real, ' + i * ', a.imag)
5 a =  1  + i * 0
6 >>> print('a jest typu', a.__class__.__name__)
7 a jest typu int
```

Notacja obiektowa: Klasa, obiekt, konkretyzacja, instancja

- **Klasa** jest abstrakcją (pewnego rodzaju przepis na obiekt) które definiuje wspólne własności (atrybuty, **pola**) oraz zachowanie (usługi, **metody**). Klasa reprezentuje **pojęcie, którego obiekt jest konkretnym wystąpieniem**.
- **Obiekt** to instancja klasy (jest konkretem, egzemplarzem klasy). Obiekt jest konkretną zmienną zbudowaną i zachowującą się zgodnie z definicją klasy.
- **Polami/atrybutami** nazywamy zmienne przechowujące dane zawarte w klasie,
- **Metody** to funkcje związane z klasą.



- **STRUKTURA KLASY** – definicja klasy

```
1 class Kalkulator():
2     pi = 3.14                       # atrybut/pole klasy
3     def dodawanie(self, a, b):      # metoda klasy
4         return(a + b)
5     def dodawanie_pi(self, a):      # metoda klasy
6         return(a + self.pi + self.dodawanie(a, a))
```

- **KONKRETYZACJA KLASY – utworzenie instancji OBIEKTU KLASY –** realizowana jest tak jak wywołanie funkcji, która zwraca obiekt klasy.

```
1 kalk = Kalkulator()   # utworzenie obiektu 1 / przestrzeń nazw "kalk"
2 kalk.pi               # wywołanie atrybutu/pola
3 kalk.dodawanie_pi(4)  # wywołanie metody
```

- Każdy obiekt jest widziany sam przez siebie pod nazwą **self**.
  - **self** używa się wewnątrz klasy, gdy potrzebne jest odwołanie obiektu do niego samego.
  - **self** jest zawsze pierwszym argumentem metody klasy, przy wywołaniu metody **self** nie ma.

Metody specjalne

Metody specjalne: `__metoda__`

- Pewne funkcje zostają wywołane w sposób automatyczny. Aby podkreślić fakt, że dana funkcja może zostać wywołana automatycznie, mają one nazwy zaczynające i kończące się dwoma podkreśleniami.
- Funkcje działające w ten sposób nazywamy **metodami specjalnymi**
- W Pythonie są różne kategorie nazw zaczynających się od podkreślenia: np. **metody specjalne** oraz**modyfikatory dostępu**, które definiują zmienne prywatne, zaczynające się od jednego lub dwóch podkreśleń, ich przeznaczeniem jest przechowanie zmiennych, do których nie należy się bezpośrednio odwoływać spoza klasy czy modułu.

- Metoda __init__ – **wywołuje się automatycznie kiedy tworzymy nową instancję obiektu**. Nie trzeba podawać nazwy tej funkcji do wywołania, interpreter wie, że ta metoda służy do konstrukcji obiektu.
- Metoda __init__ **jest konstruktorem za pomocą, którego możmy przekazać wartości argumentów**.
- Metoda ta powinna wykonywać wszystkie operacje potrzebne do zainicjowania nowego obiektu, w szczególności powinna ona **nadawać wartości jego atrybutom/polom**.
- Pod innymi względami jest ona zupełnie zwyczajna, w szczególności można ją wywołać drugi i trzeci raz podając explicite jej nazwę.

```
1  class Kalkulator():
2      pi = 3.14
3      def __init__(self):
4          self.x = 2
5          print('utworzono obiekt')
6      def dodawanie(self, a, b):
7          return(a + b)
8      def dodawanie_pi(self, a):
9          return(a + self.pi)
10
11 kalk = Kalkulator()
12 kalk.__init__()
```

```
1  utworzono obiekt
```

Lista metod specjalnych:
https://pl.python.org/docs/ref/node15.html

- Metoda `__init__` – **wywołuje się automatycznie kiedy tworzymy nową instancję obiektu**. Nie trzeba podawać nazwy tej funkcji do wywołania, interpreter wie, że ta metoda służy do konstrukcji obiektu.
- Metoda `__init__` **jest konstruktorem za pomocą, którego możmy przekazać wartości argumentów**.
- Metoda ta powinna wykonywać wszystkie operacje potrzebne do zainicjowania nowego obiektu, w szczególności powinna ona **nadawać wartości jego atrybutom/polom**.
- Pod innymi względami jest ona zupełnie zwyczajna, w szczególności można ją wywołać drugi i trzeci raz podając explicite jej nazwę.

```
1  class Kalkulator():
2      def __init__(self, a, b):
3          self.a = a
4          self.b = b
5      def dodawanie(self):
6          return(a + b)
7      def odejmowanie(self):
8          return(a - b)
9
10 kalk = Kalkulator(3, 4)
11 kalk.__init__()
```

```
1  utworzono obiekt
```

Lista metod specjalnych:
https://pl.python.org/docs/ref/node15.html

Podstawowe założenia paradygmatu obiektowego

- **Abstrakcja** (ang. *abstraction*) - czyli model, który w rzeczywistości nie reprezentuje żadnego istniejącego obiektu, ale który stanowi podstawę do definiowania obiektów. Najczęściej utożsamiany z klasą. Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, zmieniać stan lub komunikować się z innymi obiektami.
- **Hermetyzacja** (kapsułkowania, ang. *encapsulation*). - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Celem jest unikanie sytuacji, w których jeden obiekt zmienia stan wewnętrzny innych obiektów w nieoczekiwany sposób. W ramach hermetyzacji każdy obiekt prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.
- **Polimorfizm** (wielopostaciowość gr. *polymorphism*) - to metoda pozwalająca funkcji wirtualnej przyjmować różne sposoby jej realizacji. Mechanizm ten pozwala programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.
- **Dziedziczenie** (ang. *inheritance*) - przejmowanie cech/metod charakterystycznych dla innej klasy, które są z nią powiązane.

- **Abstrakcja** (ang. *abstraction*) - czyli model, który w rzeczywistości nie reprezentuje żadnego istniejącego obiektu, ale który stanowi podstawę do definiowania obiektów. Najczęściej utożsamiany z klasą. Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, zmieniać stan lub komunikować się z innymi obiektami.
- **Hermetyzacja** (kapsułkowania, ang. *encapsulation*). - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Celem jest unikanie sytuacji, w których jeden obiekt zmienia stan wewnętrzny innych obiektów w nieoczekiwany sposób. W ramach hermetyzacji każdy obiekt prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.
- **Polimorfizm** (wielopostaciowość gr. *polymorphism*) - to metoda pozwalająca funkcji wirtualnej przyjmować różne sposoby jej realizacji. Mechanizm ten pozwala programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.
- **Dziedziczenie** (ang. *inheritance*) - przejmowanie cech/metod charakterystycznych dla innej klasy, które są z nią powiązane.

- **Abstrakcja** (ang. *abstraction*) - czyli model, który w rzeczywistości nie reprezentuje żadnego istniejącego obiektu, ale który stanowi podstawę do definiowania obiektów. Najczęściej utożsamiany z klasą. Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, zmieniać stan lub komunikować się z innymi obiektami.
- **Hermetyzacja** (kapsułkowania, ang. *encapsulation*). - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Celem jest unikanie sytuacji, w których jeden obiekt zmienia stan wewnętrzny innych obiektów w nieoczekiwany sposób. W ramach hermetyzacji każdy obiekt prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.
- **Polimorfizm** (wielopostaciowość gr. *polymorphism*) - to metoda pozwalająca funkcji wirtualnej przyjmować różne sposoby jej realizacji. Mechanizm ten pozwala programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.
- **Dziedziczenie** (ang. *inheritance*) - przejmowanie cech/metod charakterystycznych dla innej klasy, które są z nią powiązane.

- **Abstrakcja** (ang. *abstraction*) - czyli model, który w rzeczywistości nie reprezentuje żadnego istniejącego obiektu, ale który stanowi podstawę do definiowania obiektów. Najczęściej utożsamiany z klasą. Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, zmieniać stan lub komunikować się z innymi obiektami.
- **Hermetyzacja** (kapsułkowania, ang. *encapsulation*). - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Celem jest unikanie sytuacji, w których jeden obiekt zmienia stan wewnętrzny innych obiektów w nieoczekiwany sposób. W ramach hermetyzacji każdy obiekt prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.
- **Polimorfizm** (wielopostaciowość gr. *polymorphism*) - to metoda pozwalająca funkcji wirtualnej przyjmować różne sposoby jej realizacji. Mechanizm ten pozwala programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.
- **Dziedziczenie** (ang. *inheritance*) - przejmowanie cech/metod charakterystycznych dla innej klasy, które są z nią powiązane.

- **Abstrakcja** (ang. *abstraction*) - czyli model, który w rzeczywistości nie reprezentuje żadnego istniejącego obiektu, ale który stanowi podstawę do definiowania obiektów. Najczęściej utożsamiany z klasą. Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, zmieniać stan lub komunikować się z innymi obiektami.
- **Hermetyzacja** (kapsułkowania, ang. *encapsulation*). - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Celem jest unikanie sytuacji, w których jeden obiekt zmienia stan wewnętrzny innych obiektów w nieoczekiwany sposób. W ramach hermetyzacji każdy obiekt prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.
- **Polimorfizm** (wielopostaciowość gr. *polymorphism*) - to metoda pozwalająca funkcji wirtualnej przyjmować różne sposoby jej realizacji. Mechanizm ten pozwala programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.
- **Dziedziczenie** (ang. *inheritance*) - przejmowanie cech/metod charakterystycznych dla innej klasy, które są z nią powiązane.

Związek pomiędzy klasami, w którym klasa przejmuje strukturę albo zachowanie zdefiniowane w innej klasie nazywamy **dziedziczeniem**. Są to procesy w których definiujemy nową klasę jako rozwinięcie istniejącej. Mechanizm dziedziczenia pozwala klasie na posiadanie wielu klas bazowych.

- Klasa – `class Osoba` – z tej klasy dziedziczą inne klasy, nazywana **klasą rodzic** (super klasą, klasą bazową ang. *Super Class, Parent Class, Base Class*).
- Klasa – `class Student(Osoba)` – jest klasą dziedzicząca nazywana jest **klasą dziecko** (podklasa ang. *Sub Class, Child Class, Derived Class*)

```
1  class Osoba:
2      """Klasa rodzic,super - klasa"""
3      def __init__(self, i, n):
4          self.imie = i
5          self.nazwisko = i
6      def get_adres(self, miasto):
7          return miasto
8
9  class Student(Osoba):
10     """Klasa dziecko, """
11     def __init__(self):
12         super().__init__()
13     def numer_indek(self, numer):
14         return numer
```

Związek pomiędzy klasami, w którym klasa przejmuje strukturę albo zachowanie zdefiniowane w innej klasie nazywamy **dziedziczeniem**. Są to procesy w których definiujemy nową klasę jako rozwinięcie istniejącej. Mechanizm dziedziczenia pozwala klasie na posiadanie wielu klas bazowych.

- Klasa – `class Osoba` – z tej klasy dziedziczą inne klasy, nazywana **klasą rodzic** (super klasą, klasą bazową ang. *Super Class, Parent Class, Base Class*).
- Klasa – `class Student(Osoba)` – jest klasą dziedzicząca nazywana jest **klasą dziecko** (podklasa ang. *Sub Class, Child Class, Derived Class*)

```
1  class Osoba:
2      """Klasa rodzic,super - klasa"""
3      def __init__(self, i, n):
4          self.imie = i
5          self.nazwisko = i
6      def get_adres(self, miasto):
7          return miasto
8
9  class Student(Osoba):
10     """Klasa dziecko, """
11     def __init__(self):
12         super().__init__()
13     def numer_indek(self, numer):
14         return numer
```

Związek pomiędzy klasami, w którym klasa przejmuje strukturę albo zachowanie zdefiniowane w innej klasie nazywamy **dziedziczeniem**. Są to procesy w których definiujemy nową klasę jako rozwinięcie istniejącej. Mechanizm dziedziczenia pozwala klasie na posiadanie wielu klas bazowych.

- Klasa – `class Osoba` – z tej klasy dziedziczą inne klasy, nazywana **klasą rodzic** (super klasą, klasą bazową ang. *Super Class, Parent Class, Base Class*).
- Klasa – `class Student(Osoba)` – jest klasą dziedzicząca nazywana jest **klasą dziecko** (podklasa ang. *Sub Class, Child Class, Derived Class*)

```
1  class Osoba:
2      """Klasa rodzic,super - klasa"""
3      def __init__(self, i, n):
4          self.imie = i
5          self.nazwisko = i
6      def get_adres(self, miasto):
7          return miasto
8
9  class Student(Osoba):
10     """Klasa dziecko, """
11     def __init__(self):
12         super().__init__()
13     def numer_indek(self, numer):
14         return numer
```

- Funkcja `super()` (bardzo przydatna) – pozwala wywołać metodę z klasy nadrzędnej.
- Jesli zdefiniowaliśmy w klasie-dziecku metodę o danej nazwie, to zastępuje ona metodę o takiej samej nazwie zdefiniowaną w klasie-rodzicu. Aby wywołać metodę z klasy-rodzic:
  - 1) podać explicite nazwę klasy rodzica (rozwiązanie prostsze),
  - 2) wykorzystać funkcję super (rozwiązanie lepsze).

```
1  class Welcome(object):
2      def hello(self):
3          return 'Hello'
4  class WarmWelcome(Welcome):
5      def hello(self):
6          return Welcome.hello(self) + "serdecznie"
7  class HeartyWelcome(Welcome):
8      def hello(self):
9          return super(HeartyWelcome, self).hello() + "gorąco !"
10 WarmWelcome().hello()  # >>> Witaj serdecznie
11 HeartyWelcome().hello() # >>> Wwitaj gorąco !
```

- Funkcja `super()` (bardzo przydatna) – pozwala wywołać metodę z klasy nadrzędnej.
- Jesli zdefiniowaliśmy w klasie-dziecku metodę o danej nazwie, to zastępuje ona metodę o takiej samej nazwie zdefiniowaną w klasie-rodzicu. Aby wywołać metodę z klasy-rodzic:
  - 1) podać explicite nazwę klasy rodzica (rozwiązanie prostsze),
  - 2) wykorzystać funkcję super (rozwiązanie lepsze).

```
1  class Welcome(object):
2      def hello(self):
3          return 'Hello'
4  class WarmWelcome(Welcome):
5      def hello(self):
6          return Welcome.hello(self) + "serdecznie"
7  class HeartyWelcome(Welcome):
8      def hello(self):
9          return super(HeartyWelcome, self).hello() + "gorąco !"
10 WarmWelcome().hello()  # >>> Witaj serdecznie
11 HeartyWelcome().hello() # >>> Wwitaj gorąco !
```

- Funkcja `super()` (bardzo przydatna) – pozwala wywołać metodę z klasy nadrzędnej.
- Jesli zdefiniowaliśmy w klasie-dziecku metodę o danej nazwie, to zastępuje ona metodę o takiej samej nazwie zdefiniowaną w klasie-rodzicu. Aby wywołać metodę z klasy-rodzic:
  - 1) podać explicite nazwę klasy rodzica (rozwiązanie prostsze),
  - 2) wykorzystać funkcję super (rozwiązanie lepsze).

```
1  class Welcome(object):
2      def hello(self):
3          return 'Hello'
4  class WarmWelcome(Welcome):
5      def hello(self):
6          return Welcome.hello(self) + "serdecznie"
7  class HeartyWelcome(Welcome):
8      def hello(self):
9          return super(HeartyWelcome, self).hello() + "gorąco !"
10 WarmWelcome().hello()  # >>> Witaj serdecznie
11 HeartyWelcome().hello() # >>> Wwitaj gorąco !
```

- Funkcja `super()` (bardzo przydatna) – pozwala wywołać metodę z klasy nadrzędnej.
- Jesli zdefiniowaliśmy w klasie-dziecku metodę o danej nazwie, to zastępuje ona metodę o takiej samej nazwie zdefiniowaną w klasie-rodzicu. Aby wywołać metodę z klasy-rodzic:
  - 1) podać explicite nazwę klasy rodzica (rozwiązanie prostsze),
  - 2) wykorzystać funkcję super (rozwiązanie lepsze).

```
1  class Welcome(object):
2      def hello(self):
3          return 'Hello'
4  class WarmWelcome(Welcome):
5      def hello(self):
6          return Welcome.hello(self) + "serdecznie"
7  class HeartyWelcome(Welcome):
8      def hello(self):
9          return super(HeartyWelcome, self).hello() + "gorąco !"
10 WarmWelcome().hello()  # >>> Witaj serdecznie
11 HeartyWelcome().hello() # >>> Wwitaj gorąco !
```

- Funkcja `super()` (bardzo przydatna) – pozwala wywołać metodę z klasy nadrzędnej.
- Jesli zdefiniowaliśmy w klasie-dziecku metodę o danej nazwie, to zastępuje ona metodę o takiej samej nazwie zdefiniowaną w klasie-rodzicu. Aby wywołać metodę z klasy-rodzic:
  - 1) podać explicite nazwę klasy rodzica (rozwiązanie prostsze),
  - 2) wykorzystać funkcję super (rozwiązanie lepsze).

```
1  class Welcome(object):
2      def hello(self):
3          return 'Hello'
4  class WarmWelcome(Welcome):
5      def hello(self):
6          return Welcome.hello(self) + "serdecznie"
7  class HeartyWelcome(Welcome):
8      def hello(self):
9          return super(HeartyWelcome, self).hello() + "gorąco !"
10 WarmWelcome().hello()  # >>> Witaj serdecznie
11 HeartyWelcome().hello() # >>> Wwitaj gorąco !
```

```python
class A():
    pass
class B():
    pass
class C(A, B): # klasa C dziedziczy po A i B
pass
a = A()
b = B()
c = C()
print('isinstance(b, B): ', isinstance(b, B)) # True
print('issubclass(B, A): ', issubclass(B, A)) # False
print('issubclass(A, B): ', issubclass(A, B)) # False
print('isinstance(c, A): ', isinstance(c, A)) # True
print('isinstance(c, B): ', isinstance(c, B)) # True
print('isinstance(c, C): ', isinstance(c, C)) # True
```

- **Polimorfizm** (z gr. wielopostaciowość) – mechanizmy pozwalające programiście używać wartości, zmiennych i podprogramów na kilka sposobów.
- **Polimorfizm to zdolność wykonywania akcji na obiekcie niezależnie od jego typu**. Zasadniczo jest to realizowane przez utworzenie klasy podstawowej i posiadanie dwóch lub więcej podklas, które wszystkie implementują metody o tej samej nazwie. Każda inna funkcja lub metoda, która manipuluje tymi obiektami, może wywoływać te same metody niezależnie od typu obiektu, na którym działa, **bez konieczności wcześniejszego sprawdzania typu**.

```
1  class Figura()
2      def oblicz_pole(self):
3          "nie ma podklasy > raise"
4          raise NotImplemented
5  class Kolo(Figura):
6      def oblicz_pole(self):
7          print("Pole koła")
8  class Kwadrat(Figura):
9      def oblicz_pole(self):
10         print("Pole kwadratu")
11
12 o1 = Kolo() # obiekt klasy
13 o2 = Kwadrat()  # obiekt klasy
14 o1.oblicz_pole()     # metoda
15 o2.oblicz_pole()     # metoda
```

- **Polimorfizm** (z gr. wielopostaciowość) – mechanizmy pozwalające programiście używać wartości, zmiennych i podprogramów na kilka sposobów.
- **Polimorfizm to zdolność wykonywania akcji na obiekcie niezależnie od jego typu**. Zasadniczo jest to realizowane przez utworzenie klasy podstawowej i posiadanie dwóch lub więcej podklas, które wszystkie implementują metody o tej samej nazwie. Każda inna funkcja lub metoda, która manipuluje tymi obiektami, może wywoływać te same metody niezależnie od typu obiektu, na którym działa, **bez konieczności wcześniejszego sprawdzania typu**.

```
1  class Figura()
2      def oblicz_pole(self):
3          "nie ma podklasy > raise"
4          raise NotImplemented
5  class Kolo(Figura):
6      def oblicz_pole(self):
7          print("Pole koła")
8  class Kwadrat(Figura):
9      def oblicz_pole(self):
10         print("Pole kwadratu")
11
12 o1 = Kolo() # obiekt klasy
13 o2 = Kwadrat()  # obiekt klasy
14 o1.oblicz_pole()     # metoda
15 o2.oblicz_pole()     # metoda
```

- **Polimorfizm** (z gr. wielopostaciowość) – mechanizmy pozwalające programiście używać wartości, zmiennych i podprogramów na kilka sposobów.
- **Polimorfizm to zdolność wykonywania akcji na obiekcie niezależnie od jego typu**. Zasadniczo jest to realizowane przez utworzenie klasy podstawowej i posiadanie dwóch lub więcej podklas, które wszystkie implementują metody o tej samej nazwie. Każda inna funkcja lub metoda, która manipuluje tymi obiektami, może wywoływać te same metody niezależnie od typu obiektu, na którym działa, **bez konieczności wcześniejszego sprawdzania typu**.

```
1  class Figura()
2      def oblicz_pole(self):
3          "nie ma podklasy > raise"
4          raise NotImplemented
5  class Kolo(Figura):
6      def oblicz_pole(self):
7          print("Pole koła")
8  class Kwadrat(Figura):
9      def oblicz_pole(self):
10         print("Pole kwadratu")
11
12 o1 = Kolo() # obiekt klasy
13 o2 = Kwadrat()  # obiekt klasy
14 o1.oblicz_pole()     # metoda
15 o2.oblicz_pole()     # metoda
```

- Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez ujawniania, w jaki sposób zaimplementowano dane cechy.
- Procesy, funkcje lub metody mogą być również abstrahowane, a kiedy tak się dzieje, konieczne są rozmaite techniki rozszerzania abstrakcji.
- Definicja klasy jest tworem abstrakcyjnym dopóki nie nastąpi konkretyzacja.

```
1  class Shape:
2      def __init__(self,color='r',filled=0):
3          self.__color = color
4          self.__filled = filled
5          def get_color(self):
6          return self.__color
7      def set_color(self, color):
8          self.__color = color
9          def get_filled(self):
10         return self.__filled
11     def set_filled(self, filled):
12         self.__filled = filled
```

- Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez ujawniania, w jaki sposób zaimplementowano dane cechy.
- Procesy, funkcje lub metody mogą być również abstrahowane, a kiedy tak się dzieje, konieczne są rozmaite techniki rozszerzania abstrakcji.
- Definicja klasy jest tworem abstrakcyjnym dopóki nie nastąpi konkretyzacja.

```
1  class Shape:
2      def __init__(self,color='r',filled=0):
3          self.__color = color
4          self.__filled = filled
5          def get_color(self):
6          return self.__color
7      def set_color(self, color):
8          self.__color = color
9          def get_filled(self):
10         return self.__filled
11     def set_filled(self, filled):
12         self.__filled = filled
```

- Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez ujawniania, w jaki sposób zaimplementowano dane cechy.
- Procesy, funkcje lub metody mogą być również abstrahowane, a kiedy tak się dzieje, konieczne są rozmaite techniki rozszerzania abstrakcji.
- Definicja klasy jest tworem abstrakcyjnym dopóki nie nastąpi konkretyzacja.

```
1  class Shape:
2      def __init__(self,color='r',filled=0):
3          self.__color = color
4          self.__filled = filled
5          def get_color(self):
6          return self.__color
7      def set_color(self, color):
8          self.__color = color
9          def get_filled(self):
10         return self.__filled
11     def set_filled(self, filled):
12         self.__filled = filled
```

- Każdy obiekt w systemie służy jako model abstrakcyjnego „wykonawcy", który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez ujawniania, w jaki sposób zaimplementowano dane cechy.
- Procesy, funkcje lub metody mogą być również abstrahowane, a kiedy tak się dzieje, konieczne są rozmaite techniki rozszerzania abstrakcji.
- Definicja klasy jest tworem abstrakcyjnym dopóki nie nastąpi konkretyzacja.

```
1  class Shape:
2      def __init__(self,color='r',filled=0):
3          self.__color = color
4          self.__filled = filled
5          def get_color(self):
6          return self.__color
7      def set_color(self, color):
8          self.__color = color
9          def get_filled(self):
10         return self.__filled
11     def set_filled(self, filled):
12         self.__filled = filled
```

- Hermetyzowanie polega na opakowaniu logiki operacji za interfejsami w taki sposób, by każda operacja była w naszym programie zapisana w kodzie tylko raz.
- W terminologii Pythona chcemy zapisać w kodzie nasze operacje na obiektach w metodach klasy, zamiast rozsiewać je po całym programie. Tak naprawdę jest to jedna z rzeczy, do których klasy świetnie się nadają -— faktoryzacja kodu w celu usunięcia jego powtarzalności i tym samym zoptymalizowania utrzymywania.
- Zaletą jest to, że zmiana działań metod pozwala na zastosowanie ich do dowolnych instancji klasy.

```
1  class Circle(Shape):
2      def __init__(self, radius):
3          super().__init__()
4          self.__radius = radius
5      def get_radius(self):
6          return self.__radius
7      def set_radius(self, radius):
8          self.__radius = radius
9      def get_area(self):
10         p=math.pi * self.__radius**2
11         return p
12 c1 = Circle(12) # obiekt klasy
13 c2 = Circle(8) # obiekt klasy
```

- Hermetyzowanie polega na opakowaniu logiki operacji za interfejsami w taki sposób, by każda operacja była w naszym programie zapisana w kodzie tylko raz.
- W terminologii Pythona chcemy zapisać w kodzie nasze operacje na obiektach w metodach klasy, zamiast rozsiewać je po całym programie. Tak naprawdę jest to jedna z rzeczy, do których klasy świetnie się nadają -— faktoryzacja kodu w celu usunięcia jego powtarzalności i tym samym zoptymalizowania utrzymywania.
- Zaletą jest to, że zmiana działań metod pozwala na zastosowanie ich do dowolnych instancji klasy.

```
1  class Circle(Shape):
2      def __init__(self, radius):
3          super().__init__()
4          self.__radius = radius
5      def get_radius(self):
6          return self.__radius
7      def set_radius(self, radius):
8          self.__radius = radius
9      def get_area(self):
10         p=math.pi * self.__radius**2
11         return p
12 c1 = Circle(12) # obiekt klasy
13 c2 = Circle(8) # obiekt klasy
```

- Hermetyzowanie polega na opakowaniu logiki operacji za interfejsami w taki sposób, by każda operacja była w naszym programie zapisana w kodzie tylko raz.
- W terminologii Pythona chcemy zapisać w kodzie nasze operacje na obiektach w metodach klasy, zamiast rozsiewać je po całym programie. Tak naprawdę jest to jedna z rzeczy, do których klasy świetnie się nadają -— faktoryzacja kodu w celu usunięcia jego powtarzalności i tym samym zoptymalizowania utrzymywania.
- Zaletą jest to, że zmiana działań metod pozwala na zastosowanie ich do dowolnych instancji klasy.

```
1  class Circle(Shape):
2      def __init__(self, radius):
3          super().__init__()
4          self.__radius = radius
5      def get_radius(self):
6          return self.__radius
7      def set_radius(self, radius):
8          self.__radius = radius
9      def get_area(self):
10         p=math.pi * self.__radius**2
11         return p
12 c1 = Circle(12) # obiekt klasy
13 c2 = Circle(8) # obiekt klasy
```

- Hermetyzowanie polega na opakowaniu logiki operacji za interfejsami w taki sposób, by każda operacja była w naszym programie zapisana w kodzie tylko raz.
- W terminologii Pythona chcemy zapisać w kodzie nasze operacje na obiektach w metodach klasy, zamiast rozsiewać je po całym programie. Tak naprawdę jest to jedna z rzeczy, do których klasy świetnie się nadają -— faktoryzacja kodu w celu usunięcia jego powtarzalności i tym samym zoptymalizowania utrzymywania.
- Zaletą jest to, że zmiana działań metod pozwala na zastosowanie ich do dowolnych instancji klasy.

```
1  class Circle(Shape):
2      def __init__(self, radius):
3          super().__init__()
4          self.__radius = radius
5      def get_radius(self):
6          return self.__radius
7      def set_radius(self, radius):
8          self.__radius = radius
9      def get_area(self):
10         p=math.pi * self.__radius**2
11         return p
12 c1 = Circle(12) # obiekt klasy
13 c2 = Circle(8) # obiekt klasy
```

Modyfikatory dostępu: public, private, protected

- Klasyczne języki obiektowe, takie jak C ++ i Java, kontrolują dostęp do zasobów klasy za pomocą publicznych, prywatnych i chronionych słów kluczowych (ang. *public*, *private*, *protected*).
- Python nie ma żadnego mechanizmu, który skutecznie ogranicza dostęp do dowolnej zmiennej lub metody instancji. Python zaleca konwencję poprzedzania nazwy zmiennej/metody pojedynczym lub podwójnym podkreśleniem, aby emulować zachowanie **modyfikatorów dostępu** np. chronionego i prywatnego czy publicznego.
- **Publiczne** (ang. *public*) – `name` – ogólnie metody zadeklarowane w klasie) są dostępne spoza klasy. Obiekt tej samej klasy jest wymagany do wywołania metody publicznej.
- **Chronione** (ang. *protected*) – `_name` – elementy klasy są dostępni wewnątrz klasy i są również dostępne dla jej podklas. Żadne inne środowisko nie ma do niego dostępu. Umożliwia to **dziedziczenie** określonych zasobów klasy nadrzędnej przez klasę podrzędną.
- **Prywatnym** (ang. *private*) – `__name` – elementom klasy odmawia się dostępu ze środowiska spoza klasy. Mogą być obsługiwane tylko z danej klasy. Takie ustawienie zmiennych prywatnych i metod publicznych zapewnia zasadę **hermetyzacja danych**.

- Klasyczne języki obiektowe, takie jak C ++ i Java, kontrolują dostęp do zasobów klasy za pomocą publicznych, prywatnych i chronionych słów kluczowych (ang. *public, private, protected*).
- Python nie ma żadnego mechanizmu, który skutecznie ogranicza dostęp do dowolnej zmiennej lub metody instancji. Python zaleca konwencję poprzedzania nazwy zmiennej/metody pojedynczym lub podwójnym podkreśleniem, aby emulować zachowanie **modyfikatorów dostępu** np. chronionego i prywatnego czy publicznego.
- **Publiczne** (ang. *public*) – `name` – ogólnie metody zadeklarowane w klasie) są dostępne spoza klasy. Obiekt tej samej klasy jest wymagany do wywołania metody publicznej.
- **Chronione** (ang. *protected*) – `_name` – elementy klasy są dostępni wewnątrz klasy i są również dostępne dla jej podklas. Żadne inne środowisko nie ma do niego dostępu. Umożliwia to **dziedziczenie** określonych zasobów klasy nadrzędnej przez klasę podrzędną.
- **Prywatnym** (ang. *private*) – `__name` – elementom klasy odmawia się dostępu ze środowiska spoza klasy. Mogą być obsługiwane tylko z danej klasy. Takie ustawienie zmiennych prywatnych i metod publicznych zapewnia zasadę **hermetyzacja danych**.

- Klasyczne języki obiektowe, takie jak C ++ i Java, kontrolują dostęp do zasobów klasy za pomocą publicznych, prywatnych i chronionych słów kluczowych (ang. *public, private, protected*).
- Python nie ma żadnego mechanizmu, który skutecznie ogranicza dostęp do dowolnej zmiennej lub metody instancji. Python zaleca konwencję poprzedzania nazwy zmiennej/metody pojedynczym lub podwójnym podkreśleniem, aby emulować zachowanie **modyfikatorów dostępu** np. chronionego i prywatnego czy publicznego.
- **Publiczne** (ang. *public*) – `name` – ogólnie metody zadeklarowane w klasie) są dostępne spoza klasy. Obiekt tej samej klasy jest wymagany do wywołania metody publicznej.
- **Chronione** (ang. *protected*) – `_name` – elementy klasy są dostępni wewnątrz klasy i są również dostępne dla jej podklas. Żadne inne środowisko nie ma do niego dostępu. Umożliwia to **dziedziczenie** określonych zasobów klasy nadrzędnej przez klasę podrzędną.
- **Prywatnym** (ang. *private*) – `__name` – elementom klasy odmawia się dostępu ze środowiska spoza klasy. Mogą być obsługiwane tylko z danej klasy. Takie ustawienie zmiennych prywatnych i metod publicznych zapewnia zasadę **hermetyzacja danych**.

- Klasyczne języki obiektowe, takie jak C ++ i Java, kontrolują dostęp do zasobów klasy za pomocą publicznych, prywatnych i chronionych słów kluczowych (ang. *public, private, protected*).
- Python nie ma żadnego mechanizmu, który skutecznie ogranicza dostęp do dowolnej zmiennej lub metody instancji. Python zaleca konwencję poprzedzania nazwy zmiennej/metody pojedynczym lub podwójnym podkreśleniem, aby emulować zachowanie **modyfikatorów dostępu** np. chronionego i prywatnego czy publicznego.
- **Publiczne** (ang. *public*) – `name` – ogólnie metody zadeklarowane w klasie) są dostępne spoza klasy. Obiekt tej samej klasy jest wymagany do wywołania metody publicznej.
- **Chronione** (ang. *protected*) – `_name` – elementy klasy są dostępni wewnątrz klasy i są również dostępne dla jej podklas. Żadne inne środowisko nie ma do niego dostępu. Umożliwia to **dziedziczenie** określonych zasobów klasy nadrzędnej przez klasę podrzędną.
- **Prywatnym** (ang. *private*) – `__name` – elementom klasy odmawia się dostępu ze środowiska spoza klasy. Mogą być obsługiwane tylko z danej klasy. Takie ustawienie zmiennych prywatnych i metod publicznych zapewnia zasadę **hermetyzacja danych**.

- Klasyczne języki obiektowe, takie jak C ++ i Java, kontrolują dostęp do zasobów klasy za pomocą publicznych, prywatnych i chronionych słów kluczowych (ang. *public, private, protected*).
- Python nie ma żadnego mechanizmu, który skutecznie ogranicza dostęp do dowolnej zmiennej lub metody instancji. Python zaleca konwencję poprzedzania nazwy zmiennej/metody pojedynczym lub podwójnym podkreśleniem, aby emulować zachowanie **modyfikatorów dostępu** np. chronionego i prywatnego czy publicznego.
- **Publiczne** (ang. *public*) – `name` – ogólnie metody zadeklarowane w klasie) są dostępne spoza klasy. Obiekt tej samej klasy jest wymagany do wywołania metody publicznej.
- **Chronione** (ang. *protected*) – `_name` – elementy klasy są dostępni wewnątrz klasy i są również dostępne dla jej podklas. Żadne inne środowisko nie ma do niego dostępu. Umożliwia to **dziedziczenie** określonych zasobów klasy nadrzędnej przez klasę podrzędną.
- **Prywatnym** (ang. *private*) – `__name` – elementom klasy odmawia się dostępu ze środowiska spoza klasy. Mogą być obsługiwane tylko z danej klasy. Takie ustawienie zmiennych prywatnych i metod publicznych zapewnia zasadę **hermetyzacja danych**.

- Klasyczne języki obiektowe, takie jak C ++ i Java, kontrolują dostęp do zasobów klasy za pomocą publicznych, prywatnych i chronionych słów kluczowych (ang. *public, private, protected*).
- Python nie ma żadnego mechanizmu, który skutecznie ogranicza dostęp do dowolnej zmiennej lub metody instancji. Python zaleca konwencję poprzedzania nazwy zmiennej/metody pojedynczym lub podwójnym podkreśleniem, aby emulować zachowanie **modyfikatorów dostępu** np. chronionego i prywatnego czy publicznego.
- **Publiczne** (ang. *public*) – `name` – ogólnie metody zadeklarowane w klasie) są dostępne spoza klasy. Obiekt tej samej klasy jest wymagany do wywołania metody publicznej.
- **Chronione** (ang. *protected*) – `_name` – elementy klasy są dostępni wewnątrz klasy i są również dostępne dla jej podklas. Żadne inne środowisko nie ma do niego dostępu. Umożliwia to **dziedziczenie** określonych zasobów klasy nadrzędnej przez klasę podrzędną.
- **Prywatnym** (ang. *private*) – `__name` – elementom klasy odmawia się dostępu ze środowiska spoza klasy. Mogą być obsługiwane tylko z danej klasy. Takie ustawienie zmiennych prywatnych i metod publicznych zapewnia zasadę **hermetyzacja danych**.

- Wszystkie elementy klasy Python są domyślnie publiczne. Dostęp do każdego elementu można uzyskać spoza środowiska klasowego.
- Publiczny charakter sprawia że nie tylko można uzyskać dostęp do atrybutów klasy, ale również można zmodyfikować ich wartości, jak pokazano poniżej.

In[3]:

```
class Employee:
    def __init__(self, name, sal):
        self.name  = name # public
        self.salary = sal # public
e1 = Employee("Alan",10000)
print('e1: ', e1.salary)
e1.salary = 2000
print('e1: ', e1.salary)
```

Out[3]:

```
e1:  10000
e1:  20000
```

- Wszystkie elementy klasy Python są domyślnie publiczne. Dostęp do każdego elementu można uzyskać spoza środowiska klasowego.
- Publiczny charakter sprawia że nie tylko można uzyskać dostęp do atrybutów klasy, ale również można zmodyfikować ich wartości, jak pokazano poniżej.

In[4]:

```
class Employee:
    def __init__(self, name, sal):
        self.name  = name # public
        self.salary = sal # public
e1 = Employee("Alan",10000)
print('e1: ', e1.salary)
e1.salary = 2000
print('e1: ', e1.salary)
```

Out[4]:

```
e1:  10000
e1:  20000
```

- Wszystkie elementy klasy Python są domyślnie publiczne. Dostęp do każdego elementu można uzyskać spoza środowiska klasowego.
- Publiczny charakter sprawia że nie tylko można uzyskać dostęp do atrybutów klasy, ale również można zmodyfikować ich wartości, jak pokazano poniżej.

In[5]:

```
class Employee:
    def __init__(self, name, sal):
        self.name  = name # public
        self.salary = sal # public
e1 = Employee("Alan",10000)
print('e1: ', e1.salary)
e1.salary = 2000
print('e1: ', e1.salary)
```

Out[5]:

```
e1:  10000
e1:  20000
```

- Konwencja Pythona, aby zabezpieczyć zmienną (**chronioną**) instancji, polega na dodaniu do niej przedrostka _ (pojedynczy znak podkreślenia). To skutecznie uniemożliwia dostęp. Menadżery podpowiedzi środowisk IDE nie widzą tych zmiennych.
- Python umożliwia dostęp i modyfikację zmiennych chronionych, jednak odpowiedzialny programista nie skorzysta z tej opcji.

In[6]:

```
class Employee:
    def __init__(self, name, sal):
        self._name = name #
        self._salary = sal
e1 = Employee("Alan",10000)
print('e1: ', e1.salary)
e1._salary = 2000
print('e1: ', e1._salary)
```

Out[6]:

```
e1:  10000
e1:  20000
```

- Konwencja Pythona, aby zabezpieczyć zmienną (**chronioną**) instancji, polega na dodaniu do niej przedrostka _ (pojedynczy znak podkreślenia). To skutecznie uniemożliwia dostęp. Menadżery podpowiedzi środowisk IDE nie widzą tych zmiennych.
- Python umożliwia dostęp i modyfikację zmiennych chronionych, jednak odpowiedzialny programista nie skorzysta z tej opcji.

In[7]:

```
class Employee:
    def __init__(self, name, sal):
        self._name = name #
        self._salary = sal
e1 = Employee("Alan",10000)
print('e1: ', e1.salary)
e1._salary = 2000
print('e1: ', e1._salary)
```

Out[7]:

```
e1:  10000
e1:  20000
```

- Konwencja Pythona, aby zabezpieczyć zmienną (**chronioną**) instancji, polega na dodaniu do niej przedrostka _ (pojedynczy znak podkreślenia). To skutecznie uniemożliwia dostęp. Menadżery podpowiedzi środowisk IDE nie widzą tych zmiennych.
- Python umożliwia dostęp i modyfikację zmiennych chronionych, jednak odpowiedzialny programista nie skorzysta z tej opcji.

In[8]:

```
class Employee:
    def __init__(self, name, sal):
        self._name = name #
        self._salary = sal
e1 = Employee("Alan",10000)
print('e1: ', e1.salary)
e1._salary = 2000
print('e1: ', e1._salary)
```

Out[8]:

```
e1:  10000
e1:  20000
```

- Podwójne podkreślenie (__) przed zmienną czyni ją **prywatną**. Należy nie "dotykać" tego elementu spoza klasy, każda próba edycji zmiennej prywatnej wywoła: `AttributeError`

In[9]:

```
class Employee:
    def __init__(self, name, sal):
        self.__name = name #
        self.__salary = sal
        e1 = Employee("Alan",10000)
        e1.__salary
```

Out[9]:

```
AttributeError: 'employee' object has no attribute '__salary'
```

- Python pozwala dokonywać modyfikacji zmiennych prywatnych, tylko wtedy jesli jest to konieczne, ale należy się powstrzymać od tego typu praktyki. Modyfikację i dostęp do zmiennej prywatnej realizujemy: `_object._class__variable`.

In[10]:

```
e1 = Employee("Alan",10000)
e1._Employee__salary = 20000
```

- Podwójne podkreślenie (__) przed zmienną czyni ją **prywatną**. Należy nie "dotykać" tego elementu spoza klasy, każda próba edycji zmiennej prywatnej wywoła: AttributeError

In[11]:

```
class Employee:
    def __init__(self, name, sal):
        self.__name = name #
        self.__salary = sal
        e1 = Employee("Alan",10000)
        e1.__salary
```

Out[11]:

```
AttributeError: 'employee' object has no attribute '__salary'
```

- Python pozwala dokonywać modyfikacji zmiennych prywatnych, tylko wtedy jesli jest to konieczne, ale należy się powstrzymać od tego typu praktyki. Modyfikację i dostęp do zmiennej prywatnej realizujemy: _object._class__variable.

In[12]:

```
e1 = Employee("Alan",10000)
e1._Employee__salary = 20000
```

- Podwójne podkreślenie (__) przed zmienną czyni ją **prywatną**. Należy nie "dotykać" tego elementu spoza klasy, każda próba edycji zmiennej prywatnej wywoła: `AttributeError`

In[13]:

```
class Employee:
    def __init__(self, name, sal):
        self.__name = name #
        self.__salary = sal
        e1 = Employee("Alan",10000)
        e1.__salary
```

Out[13]:

```
AttributeError: 'employee' object has no attribute '__salary'
```

- Python pozwala dokonywać modyfikacji zmiennych prywatnych, tylko wtedy jesli jest to konieczne, ale należy się powstrzymać od tego typu praktyki. Modyfikację i dostęp do zmiennej prywatnej realizujemy: `_object._class__variable`.

In[14]:

```
e1 = Employee("Alan",10000)
e1._Employee__salary = 20000
```

Podsumowanie – terminologia OPP

Podsumowanie koncepcji związanych z programowaniem zorientowanym obiektowo:

- **Klasa** (ang. *class*) - definiuje wspólne własności i zachowanie obiektów (**Klasa to złączenie danych i funkcji w jeden obiekt**). Klasa zawiera funkcje przeznaczone do używania z obiektami tej klasy - metody. Klasa jest opisem, definicją. Obiekt jest konkretną zmienną zbudowaną i zachowującą się zgodnie z definicją klasy. Na najprostrzym poziomie klasa jest po prostu przestrzenią nazw (namespaces)
- **Obiekt** (ang. *object*) - są instancjami klas. Zmienne przechowujące dane zawarte w obiekcie nazywa się **polami**, natomiast funkcje związane z obiektem, nazywa się **metodami**. Konkretyzacja klasy tworzy obiekt. Obiekt jest instancją klasy
- Instancje (ang. *instance*) - są specyficznymi realizacjami obiektu, są tworzone zgodnie z definicją podaną w klasie. Innymi słowy, instancje są danymi, które należą do indywidualnego obiektu, każdy obiekt ma ich własną kopię
- Konkretyzacja (ang. *instantiation*) - utworzenie instancji tj. exemplarza obiektu.
- Zmienne klasy - należą do całości klasy, jest tylko jedna kopia każdej zmiennej.

- Instancje zmiennych lub atrybutów - dane, które należą do indywidualnego obiektu, każdy obiekt ma ich własną kopię
- Zmienne składowe - odwołują się zarówno do klas oraz zmiennych instancji, które są zdefiniowane przez konkretne klasy
- Metody w klasach - należą do całości klas I mają dostęp tylko do zmiennych klas oraz wprowadzanych w wywoływaniu procedur
- Metody instancji - należą do indywidualnych obiektów, mają dostęp do zmiennych instancji dla specyficznych obiektów, dla których są wywoływane, wprowadzanych oraz zmiennych kla
- **Dziedziczenie** (ang. *inheritance*) – w Pythonie oparte na wyszukiwaniu atrybutów. Przejmowanie cech charakterystycznych dla innej klasy, które są z nią powiązane. Instancja:

- **Polimorfizm** – Sytuację kiedy obiekty dwóch różnych klas mają wspólny zestaw metod i pól i można je używać zamiennie. Mechanizmy pozwalające programiście używać wartości, zmiennych i podprogramów na kilka różnych sposobów. Innymi słowy, znaczenie metody (operacji) uzależnione jest od typu (klasy) obiektu, na którym się tą operację wykonuje.
- **Duck-typing** – inaczej **leniwy polimorfizm**, oznacza, że definiujemy klasy które niekoniecznie mają wspólnego przodka, a jedynie definiują wszystkie atrybuty potrzebne do wykonania danej operacji.
- **Hermetyzacja** (enkapsulacja) - ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Można modyfikować implementację interfejsu bez wpływania na użytkowników tego obiektu.
- **Kompozycja** - osadzanie innych obiektów w obiekcie pojemnika.
- **Wzorce projektowe** (ang. design patterns) - często wykorzystywane struktury programowania zorientowanego obiektowo.

Podsumowanie koncepcji związanych z programowaniem zorientowanym obiektowo:

- Kompleksowe omówienie OPP w książce: (Lutz, 2011, Rozdział 26 i 27)
- https://brain.fuw.edu.pl/edu/index.php/TI/Wst%C4%99p_do_programowania_obiektowego
- https://pl.python.org/docs/tut/node11.html

M. Lutz. *Python. Wprowadzenie*. Helion, 2011.

Dziękuje za uwagę

Kinga Węzka kinga.wezka@pw.edu.pl