1. Programowanie obiektowe – to nowe podejście do pisania programów, które ma ułatwić zrozumienie kodu. W kodzie bowiem korzystamy z klas i ich obiektów, co odzwierciedla rzeczywistość - mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji.

Programowanie imperatywne - to paradygmat programowania, w którym skupiamy się na tym co chcemy zrobić (wykonujemy konkretne czynności step-by-step). Tworzymy wynik (języki III poziomu).

Programowanie funkcyjne - podejście, w którym skupiamy się na tym, co chcemy osiągnąć (SQL języki iV poziomu).

1. To metoda magiczna automatycznie uruchamiana przy tworzeniu nowych obiektów klasy
2. Mówi on interpreterowi, że metoda należy do danej klasy. Ponadto self umożliwia odwołania się do pól danej klasy.
3. class Auto:

def \_\_init\_\_(self, marka\_: str, model\_: str) -> None:

self.marka = marka\_

self.model = model\_

def description(self) -> None:

print(f’To auto to model {self.model} marki {self.marka}’)

def main():

samochod = Auto(‘Fiat’, ‘Panda’)

samochod.description()

if \_\_name\_\_ == “\_\_main\_\_”:

main()

1. Jest to rodzaj hermetyzowania danych przed dostępem z „zewnątrz” klasy. Czyli żeby zmodyfikować wartość pola nie możemy odwołać się do niego bezpośrednio ale mozemy stworzyć dedykowaną do tego metodę.
2. Do pola prywatnego nie mamy dostępu poza klasą i atrybuty prywatne nie są dziedziczone.
3. Przy pomocy getterów i setterów.
4. class Auto:

def \_\_init\_\_(self, marka\_: str, model\_: str) -> None:

self.\_\_marka = marka\_

self.\_model = model\_

def description(self) -> None:

print(f’To auto to model {self.\_model} marki {self.\_\_marka}’)

def set\_marka(self, nowa\_marka\_: str) -> None:

self.\_\_marka = nowa\_marka\_

def main():

samochod = Auto(‘Fiat’, ‘Panda’)

samochod.description()

samochod.\_\_marka = ‘Audi” #BŁĄD!

samochod.set\_marka(‘Audi’)

if \_\_name\_\_ == “\_\_main\_\_”:

main()

1. Dziedziczenie umożliwia przekazywanie pewnych cech klasy bazowej do klas potomnych.
2. class Wielokat:

def \_\_init\_\_(self, boki\_: list[int]):

self.boki = boki\_

def oblicz\_obwod(self):

print(f’Obwód wynosi: {sum(self.boki)\*2}’)

class Prostokat(Wielokat):

def \_\_init\_\_(self, bok1\_, bok2\_):

super().\_\_init\_\_([bok1\_, bok2\_])

def oblicz\_pole(self):

print(f’Pole wynosi: {sum(self.boki)\*2}’)

def main():

prostokat = Prostokat(2, 5)

prostokat.oblicz\_obwod()

prostokat.oblicz\_pole()

if \_\_name\_\_ == “\_\_main\_\_”:

main()

1. Method resolution order – kolejność wykonywania metod dziedziczonych

class Parent1:

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        print('In parent1')

class Parent2:

    def \_\_init\_\_(self) -> None:

        print('In parent2')

class Child(Parent1, Parent2):

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        print('in child class')

def main():

    child = Child()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

1. Nadawanie nowych zachowań znanym nam operatorom

class Uzytkownik:

    def \_\_init\_\_(self, name\_: str) -> None:

        self.name = name\_

    def \_\_add\_\_(self, obj) -> None:

        print(f'{self.name}, {obj.name}')

def main():

    user1 = Uzytkownik('Mark')

    user2 = Uzytkownik('Joe')

    user1+user2

if \_\_name\_\_ == '\_\_main\_\_':

    main()