

Wstęp do obiektowości

Klasy i dziedziczenie

Klasy - Wprowadzenie

Dotychczas
koncentrowaliśmy
się na zmiennych
oraz funkcjach.

Stwórzmy funkcję,
która wypisze podstawowe
informacje o startującym
samochodzie.

```
s1_kolor = 'czerwony'
```

```
s1_marka = 'fiat'
```

```
s2_kolor = 'czarny'
```

```
s2_marka = 'bmw'
```

```
def run(kolor, marka):  
    print("Startuje", kolor, marka)
```

```
run(s1_kolor, s1_marka)
```

```
run(s2_kolor, s2_marka)
```



Wstęp do obiektowości

Klasy - Wprowadzenie

Jakie problemy możemy dostrzec w przytoczonym kodzie źródłowym?

```
s1_kolor = 'czerwony'
```

```
s1_marka = 'fiat'
```

```
s2_kolor = 'czarny'
```

```
s2_marka = 'bmw'
```

```
def run(kolor, marka):  
    print("Startuje", kolor, marka)
```

```
run(s1_kolor, s1_marka)
```

```
run(s2_kolor, s2_marka)
```



Wstęp do obiektowości

Klasy - Wprowadzenie

Jakie problemy możemy dostrzec w przytoczonym kodzie źródłowym?

1. Możliwa błędna kolejność podanych argumentów w funkcji run.

```
s1_kolor = 'czerwony'  
s1_marka = 'fiat'
```

```
s2_kolor = 'czarny'  
s2_marka = 'bmw'
```

```
def run(kolor, marka):  
    print("Startuje", kolor, marka)
```

```
run(s1_kolor, s1_marka)  
run(s2_marka, s2_kolor)
```



Wstęp do obiektowości

Klasy - Wprowadzenie

Jakie problemy możemy dostrzec w przytoczonym kodzie źródłowym?

2. “Zmieszanie” dwóch różnych samochodów (np. kolor z s1, marka z s2).

```
s1_kolor = 'czerwony'  
s1_marka = 'fiat'
```

```
s2_kolor = 'czarny'  
s2_marka = 'bmw'
```

```
def run(kolor, marka):  
    print("Startuje", kolor, marka)
```

```
run(s1_kolor, s1_marka)  
run(s1_kolor, s2_marka)
```



Wstęp do obiektowości

Klasy - Wprowadzenie

Jakie problemy możemy dostrzec w przytoczonym kodzie źródłowym?

3. Podanie dwa razy tych samych argumentów (np. marka zamiast koloru).

```
s1_kolor = 'czerwony'  
s1_marka = 'fiat'
```

```
s2_kolor = 'czarny'  
s2_marka = 'bmw'
```

```
def run(kolor, marka):  
    print("Startuje", kolor, marka)
```

```
run(s1_kolor, s1_marka)  
run(s2_marka, s2_marka)
```



Wstęp do obiektowości

Klasy - Wprowadzenie

Jakie problemy możemy dostrzec w przytoczonym kodzie źródłowym?

Główny problem:

Poszczególne zmienne określające dany samochód nie są ze sobą w żaden sposób powiązane.

```
s1_kolor = 'czerwony'  
s1_marka = 'fiat'
```

```
s2_kolor = 'czarny'  
s2_marka = 'bmw'
```

```
def run(kolor, marka):  
    print("Startuje", kolor, marka)
```

```
run(s1_kolor, s1_marka)  
run(s2_marka, s2_marka)
```



Wstęp do obiektowości

Klasy - Definicja klasy

Korzystając z klas możemy powiązać poszczególne zmienne ze sobą.

Definicja klasy (klasa)
określa zachowanie i stan obiektu.

Na podstawie definicji klasy tworzone są konkretne obiekty - instancje klasy.

```
class Samochod:  
    pass
```

definicja klasy



```
def run(sam):  
    print("Startuje", sam.kolor, sam.marka)
```

```
s1 = Samochod()  
s1.kolor = 'czerwony'  
s1.marka = 'fiat'
```

instancja klasy



```
run(s1)
```



Wstęp do obiektowości

Klasy - Metody klasy

Co możemy poprawić
w omawianym przykładzie?

```
class Samochod:  
    pass  
  
def run(sam):  
    print("Startuje", sam.kolor, sam.marka)  
  
s1 = Samochod()  
s1.kolor = 'czerwony'  
s1.marka = 'fiat'  
  
run(s1)
```



Wstęp do obiektowości

Klasy - Metody klasy

Co możemy poprawić
w omawianym przykładzie?

Warto napisać dodatkową
funkcję, która jako parametr
będzie przyjmowała obiekt
samochodu oraz atrybuty,
które należy ustawić
w przekazanym obiekcie.
Stosując takie rozwiązanie
nie pominiemy żadnego
z atrybutów.

```
class Samochod:  
    pass
```

```
def run(sam):  
    print("Startuje", sam.kolor, sam.marka)
```

```
s1 = Samochod()  
s1.kolor = 'czerwony'  
s1.marka = 'fiat'
```

```
run(s1)
```

**co jeśli zapomnimy
ustawić, któregoś
z parametrów?**



Wstęp do obiektowości

Klasy - Metody klasy

Nowa funkcja `init()`
ustawia odpowiednie
zmienne w **konkretnym**
obiekcie samochodu.

Powiązaliśmy ze sobą
zmienne, czy to samo
możemy zrobić z funkcjami?

Co się stanie jeśli wywołamy
którąś z funkcji na obiekcie
innej klasy niż samochód?

```
class Samochod:
    pass

def run(sam):
    print("Startuje", sam.kolor, sam.marka)

def init(sam, kolor, marka):
    sam.kolor = kolor
    sam.marka = marka

s1 = Samochod()
init(s1, 'czerwony', 'fiat')

run(s1)
```




Wstęp do obiektowości

Klasy - Metody klasy

Funkcje związane z daną klasą możemy umieścić w jej definicji.

Zwróćmy uwagę, że jedyny parametr funkcji zmienił swoją nazwę z 'sam' na 'self', wskazuje on instancję klasy.

Nazwa tego parametru jest dowolna, jednak PEP 8 zaleca stosowanie nazwy 'self'.



```
class Samochod:
    def run(self):
        print("Startuje", self.kolor, self.marka)
```

```
def init(sam, kolor, marka):
    sam.kolor = kolor
    sam.marka = marka
```

```
s1 = Samochod()
init(s1, 'czerwony', 'fiat')
```

```
s1.run()  # --- wcześniej ---> run(s1)
```



Wstęp do obiektowości

Klasy - Metody klasy

Zmienił się także sposób wywołania funkcji `run()`. Pierwszy parametr funkcji znajduje się teraz z lewej strony.

```
class Samochod:
    def run(self):
        print("Startuje", self.kolor, self.marka)

    def init(sam, kolor, marka):
        sam.kolor = kolor
        sam.marka = marka

s1 = Samochod()
init(s1, 'czerwony', 'fiat')

s1.run()  # --- wcześniej ---> run(s1)
```



Klasy - Metody klasy


Również funkcję `__init__()` możemy umieścić w definicji klasy. Funkcję taką nazywamy konstruktorem lub po prostu funkcją `init`. Ustawia ona początkowy stan obiektu.

Również w przypadku tej funkcji zmieniła się nazwa pierwszego parametru i sposób jej wywołania.

```
class Samochod:
    def __init__(self, kolor, marka):
        self.kolor = kolor
        self.marka = marka

    def run(self):
        print("Startuje", self.kolor, self.marka)

s1 = Samochod("czerwony", "fiat")
s1.run()
```



Wstęp do obiektowości

Dziedziczenie - Wprowadzenie

Zmodyfikujmy lekko nasz przykład. Uwzględnijmy poziom paliwa, jakim dysponuje samochód i uzależnijmy od tego jego start.

Dla samochodu benzynowego stwórzmy klasę SamochodBenzyna.

```
class SamochodBenzyna:
    def __init__(self, kolor, marka, benzyna):
        self.kolor = kolor
        self.marka = marka
        self.benzyna = benzyna

    def run(self):
        if self.czy_mozna_wystartowac():
            print("Startuje",
                  self.kolor, self.marka)

    def czy_mozna_wystartowac(self):
        return self.benzyna > 10
```

nowe pole →

→ **uzależnienie startu od poziomu benzyny w baku**



Dziedziczenie - Wprowadzenie

Podobnie sytuacja będzie wyglądała dla samochodu na gaz. Stwórzmy klasę SamochodLPG z odpowiednimi polami, czyli zmiennymi klasy.

```
class SamochodLPG:
    def __init__(self, kolor, marka, lpg):
        self.kolor = kolor
        self.marka = marka
        self.lpg = lpg

    def run(self):
        if self.czy_mozna_wystartowac():
            print("Startuje",
                  self.kolor, self.marka)

    def czy_mozna_wystartowac(self):
        return self.lpg > 20
```



Dziedziczenie - Wprowadzenie

Czy możemy zrobić to lepiej?

```
class SamochodLPG:
    def __init__(self, kolor, marka, lpg):
        self.kolor = kolor
        self.marka = marka
        self.lpg = lpg

    def run(self):
        if self.czy_mozna_wystartowac():
            print("Startuje",
                  self.kolor, self.marka)

    def czy_mozna_wystartowac(self):
        return self.lpg > 20
```



Wstęp do obiektowości

Dziedziczenie - Hierarchia klas

Pomysł:

Stwórzmy jedną ogólną
definicję klasy

samochód - uniwersalną
dla każdego samochodu.

Ogólna wersja nie zawiera pól
i metod odpowiedzialnych
za poziom paliwa.

klasa nadrzędna/bazowa

```
class Samochod:
    def __init__(self, kolor, marka):
        self.kolor = kolor
        self.marka = marka

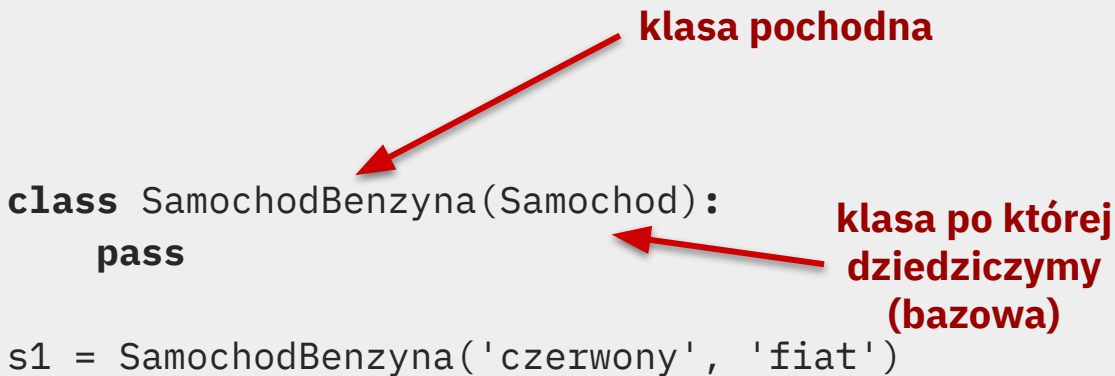
    def run(self):
        if self.czy_mozna_wystartowac():
            print("Startuje",
                  self.kolor, self.marka)
```



Dziedziczenie - Hierarchia klas

Na podstawie klasy
Samochod stworzymy klasę
SamochodBenzyna
przeznaczoną
dla samochodów
benzynowych.

**Klasa ta będzie zawierała
wszystkie pola i metody
klasy Samochód.
Mechanizm ten nazywamy
dziedziczeniem.**



```
class SamochodBenzyna(Samochod):  
    pass  
  
s1 = SamochodBenzyna('czerwony', 'fiat')
```

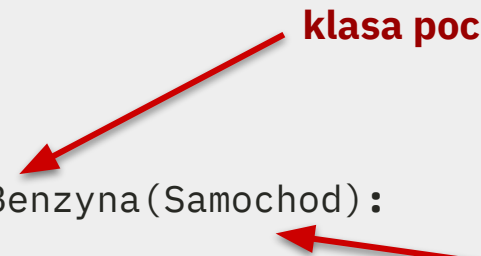
klasa pochodna

**klasa po której
dziedziczymy
(bazowa)**



Dziedziczenie - Hierarchia klas

Klasę po której dziedziczymy nazywamy klasą bazową, nowo tworzoną klasę nazywamy klasą pochodną.



```
class SamochodBenzyna(Samochod):  
    pass  
  
s1 = SamochodBenzyna('czerwony', 'fiat')
```

klasa pochodna

klasa po której dziedziczymy (bazowa)



Dziedziczenie - Hierarchia klas

Obiekt utworzony na podstawie klasy `SamochodBenzyna` nie jest w pełni funkcjonalny.

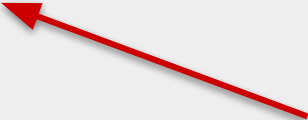
Zawiera on co prawda metodę `run()`, jednak nigdzie nie została zdefiniowana metoda

`czy_mozna_wystartowac()`.

Próba wywołania metody `run()` spowoduje błąd wykonania.

```
class SamochodBenzyna(Samochod):  
    pass
```

```
s1 = SamochodBenzyna('czerwony', 'fiat')  
s1.run()
```



**BŁĄD! brak metody
czy_mozna_wystartowac**



Wstęp do obiektowości

Dziedziczenie - Hierarchia klas

Uzupełnijmy definicję klasy
SamochodBenzyna o potrzebne
pola i metody.

Tworzymy nowy konstruktor
(funkcję `__init__()`)
uwzględniający pole benzyna,
a także nową funkcję
`czy_mozna_wystartowac()`.

Co można zrobić lepiej?

```
class SamochodBenzyna(Samochod):  
    def __init__(self, kolor, marka, benzyna):  
        self.kolor = kolor  
        self.marka = marka  
        self.benzyna = benzyna  
  
    def czy_mozna_wystartowac(self):  
        return self.benzyna > 10
```

```
s1 = SamochodBenzyna('czerwony', 'fiat', 15)  
s1.run()
```



Wstęp do obiektowości

Dziedziczenie - Hierarchia klas

Zasada DRY - Don't Repeat Yourself!

Istniał już konstruktor, który potrafił ustawić pola kolor oraz marka.

Zamiast na nowo pisać kolejne instrukcje przypisania skorzystajmy z konstruktora dostępnego w klasie Samochod, zrobimy to przy pomocy funkcji **super()**.

```
class SamochodBenzyna(Samochod):  
    def __init__(self, kolor, marka, benzyna):  
        super().__init__(kolor, marka)  
        self.benzyna = benzyna  
  
    def czy_mozna_wystartowac(self):  
        return self.benzyna > 10  
  
s1 = SamochodBenzyna('czerwony', 'fiat', 15)  
s1.run()
```



Wstęp do obiektowości

Dziedziczenie - Widoczność atrybutów

Tworząc klasy musimy zadbać o widoczność pól i metod.

Nie zawsze chcemy, by każde pole było dostępne do użytku poza metodami danej klasy.

Zachowanie to nazywamy **hermetyzacją** lub inaczej **enkapsulacją**.

Pole prywatne - dostępne tylko dla metod danej klasy, tworzymy poprzez **dodanie znaku podkreślenia na początku nazwy**.

```
class Samochod:
    def __init__(self, kolor, marka):
        self.kolor = kolor
        self.marka = marka

    def run(self):
        if self._czy_mozna_wystartowac():
            print("Startuje",
                  self.kolor, self.marka)
```

pole publiczne

wywołanie metody prywatnej

C:


Wstęp do obiektowości

Dziedziczenie - Widoczność atrybutów

Pola prywatne utworzone za pomocą jednego znaku podkreślenia są prywatne w rozumieniu umowy między programistami, określa to dokument PEP 8.

Interpreter nie weryfikuje dostępu do pola/metody.

```
class SamochodBenzyna(Samochod):  
    def __init__(self, kolor, marka, benzyna):  
        super().__init__(kolor, marka)  
        self._benzyna = benzyna  
  
    def _czy_mozna_wystartowac(self):  
        return self._benzyna > 10
```

 **pole prywatne**

```
s1 = SamochodBenzyna('czerwony', 'fiat', 15)  
s1.run()
```



Dziedziczenie - Widoczność atrybutów

Pola/metody prywatne można również utworzyć za pomocą **podwójnego znaku podkreślenia**. W takim przypadku **interpreter weryfikuje dostęp i powiadamia o błędzie**. Do takich pól/metod nie można odwołać się w klasie nadrzędnej (bazowej) i pochodnej.

Nazwy zaczynające się i kończące się dwoma znakami podkreślenia są funkcjami o specjalnym przeznaczeniu (przykład: funkcja `__init__()`).



Dziedziczenie - Atrybuty statyczne

W definicji klasy możemy również umieścić pola i metody statyczne.

Pole statyczne przyjmuje jedną wartość dla wszystkich instancji danej klasy.

Pola/metody statyczne nie muszą być wywoływane na konkretnym obiekcie danej klasy.

```
class Samochod:
    stala_mph_kph = 1.6093

    # ...

    @staticmethod
    def zamienMphNaKph(mile):
        return mile * Samochod.stala_mph_kph

kph = Samochod.zamienMphNaKph(1)
print('kph =', kph)
```

pole statyczne

funkcja statyczna



Pytania

1. Co określa definicja klasy?
2. Czym jest parametr `self`, za co odpowiada?
3. Za co odpowiada konstruktor?
4. Co nazywamy dziedziczeniem?
5. Czym jest hermetyzacja?
6. W jaki sposób tworzymy pola/metody prywatne?



Literatura

1. A First Look at Classes,
<https://docs.python.org/3/tutorial/classes.html#a-first-look-at-classes>
2. Private Variables,
<https://docs.python.org/3/tutorial/classes.html#private-variables>
3. Inheritance,
<https://docs.python.org/3/tutorial/classes.html#inheritance>
4. Staticmethod,
<https://www.programiz.com/python-programming/methods/built-in/staticmethod>



