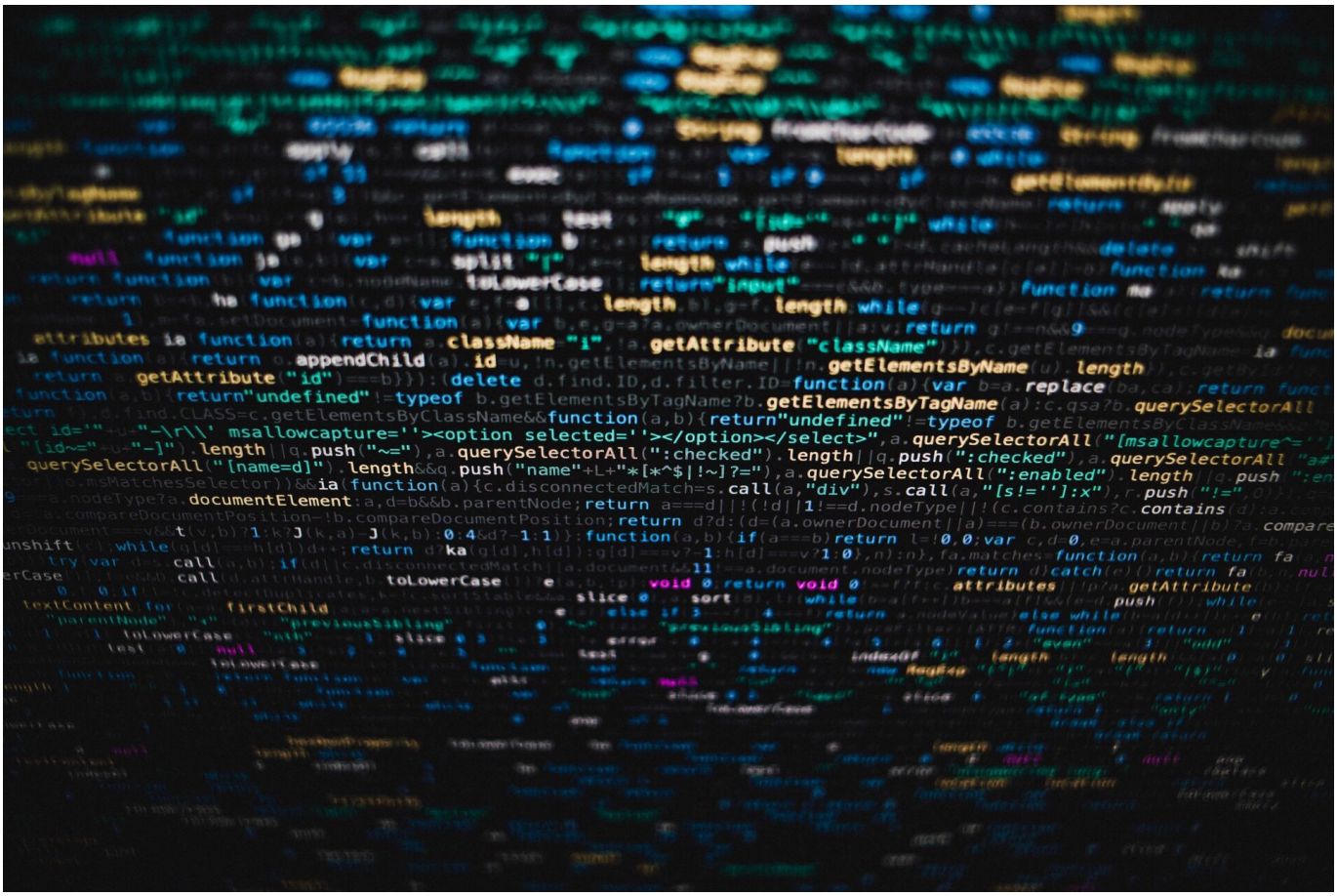

Zasady programowania obiektowego w języku Python

- Wprowadzenie
- Przeczytaj
- Prezentacja multimedialna
- Sprawdź się
- Dla nauczyciela

Źródło: domena publiczna.

Wiemy już, że programowanie obiektowe opiera się na kilku zasadach. Czym są abstrakcja, dziedziczenie, polimorfizm i hermetyzacja dowiedzieliśmy się w e-materiale Zasady programowania obiektowego.

W tym e-materiale zajmiemy się realizacją zasad programowania obiektowego w języku Python.

Ciekawi cię, jak wyglądają implementacje w innych językach programowania? Możesz się z nimi zapoznać w dwóch pozostałych e-materiałach z tej serii:

- Zasady programowania obiektowego w języku C++,
- Zasady programowania obiektowego w języku Java.

Twoje cele

- Prześledzisz zaawansowane zasady programowania obiektowego w języku Python.
- Zaimplementujesz definicje klas wykorzystujących techniki programowania obiektowego.
- Stworzysz program wykorzystujący techniki programowania obiektowego i definicje klas.

Przeczytaj

Hermetyzacja

Hermetyzacja oznacza ukrywanie implementacji. W języku `Python` polega ona na oznaczeniu wybranych metod bądź pól, które nie powinny być używane poza klasą, w której są zdefiniowane.

Przykład 1

Przeanalizujmy przypadek, w którym tworzymy oprogramowanie sterujące silnikiem okrętu. Chcemy umożliwić sterowanie w taki sposób, aby przy ustawieniach: `cała wstecz`, `silniki stop`, `cała naprzód` niemożliwa była zmiana z pozycji `cała naprzód` do `cała wstecz` i odwrotnie – z pominięciem `silniki stop`. Dla celów wizualizacji użyjemy wypisywania danych na ekranie, z zastosowaniem techniki f-string.

Napiszemy definicję klasy bez wykorzystania hermetyzacji.

```
class SilnikOkretowy:
    stany = ["Cała wstecz", "Silniki STOP", "Cała naprzód"]

    def __init__(self):
        self.wskazanie = 1
        self.stan_silnika = self.stany[self.wskazanie]
        self.aktualizuj_silniki(0)

    def aktualizuj_silniki(self, krok):
        if krok < 0 and self.wskazanie > 0:
            print("Zmiana silników")
            self.wskazanie += krok
        if krok > 0 and self.wskazanie < 2:
            print("Zmiana silników")
            self.wskazanie += krok

        self.stan_silnika = self.stany[self.wskazanie]
        print(f"Aktualnie silniki: {self.stan_silnika}")

    def silniki_naprzod(self):
        self.aktualizuj_silniki(1)

    def silniki_wstecz(self):
```

```
        self.aktualizuj_silniki(-1)


# inicjalizacja zmiennej obiektowej
batory = SilnikOkretowy()
# Aktualnie silniki: Silniki STOP

batory.silniki_naprzod()
# Zmiana silników
# Aktualnie silniki: Cała naprzód

batory.silniki_naprzod()
# Aktualnie silniki: Cała naprzód

batory.silniki_naprzod()
# Aktualnie silniki: Cała naprzód

batory.silniki_wstecz()
# Zmiana silników
# Aktualnie silniki: Silniki STOP

batory.silniki_wstecz()
# Zmiana silników
# Aktualnie silniki: Cała wstecz

batory.silniki_wstecz()
# Aktualnie silniki: Cała wstecz
```

Definiowanie bez żadnych dodatkowych oznaczeń pól czy metod, które powinny być prywatne, może skończyć się błędami w kodzie. Np. ktoś nie wiedząc o tym, że nie powinien modyfikować jakiejś zmiennej, nadpisze jej wartość, co spowoduje błąd – jak w przedstawionym przykładzie:

```
print(batory.stan_silnika)
# Cała wstecz
print(batory.wskazanie)
# 0

batory.wskazanie = 2
print(batory.wskazanie)
# 2
```

```
print(batory.stan_silnika)
# Cała wstecz
batory.aktualizuj_silniki(0)
# Aktualnie silniki: Cała naprzód

# wskazanie jest zmienną która powinna być modyfikowana tylko
# przez przeznaczone do tego funkcje, ręczna zmiana jak tu może
# doprowadzić do tego, że będziemy się odwoływać do indeksu
# większego niż rozmiar tablicy i zostanie zwrócony błąd
batory.wskazanie = 5
batory.aktualizuj_silniki(0)
# Traceback (most recent call last):
#   File "<pyshell#55>", line 1, in <module>
#     batory.aktualizuj_silniki(0)
#   File "/home/python/idle-src/0373_silniki_okr_hermet.py", li
#     self.stan_silnika = self.stany[self.wskazanie]
# IndexError: list index out of range
```

Hermetyzacja w języku Python polega na umowie między programistami, że będą przestrzegać ogólnie przyjętych norm. Metody i pola, które chcemy oznaczyć jako prywatne (czyli takie, które nie powinny być używane poza daną klasą), oznaczamy znakiem "`_`" na początku nazwy zmiennej lub metody np. `_x`. Środowiska programistyczne często informują użytkownika, gdy w złym miejscu skorzysta ze zmiennej nazwanej w ten sposób.

```
class SilnikOkretowyPrywatny:
    # zmiana tablicy na krotkę, ponieważ jest ona niezmienna (i
    stany = ("Cała wstecz", "Silniki STOP", "Cała naprzód")

    def __init__(self):
        self._wskazanie = 1
        self._stan_silnika = self.stany[self._wskazanie]
        self._aktualizuj_silniki(0)

    def _aktualizuj_silniki(self, krok):
        if krok < 0 and self._wskazanie > 0:
            print("Zmiana silników")
            self._wskazanie += krok
        if krok > 0 and self._wskazanie < 2:
            print("Zmiana silników")
            self._wskazanie += krok
```

```
17
18          self._stan_silnika = self.stany[self._wskazanie]
19          print(f"Aktualnie silniki: {self._stan_silnika}")
20
21      def silniki_naprzod(self):
22          self._aktualizuj_silniki(1)
23
24      def silniki_wstecz(self):
25          self._aktualizuj_silniki(-1)
```

Przy takim zapisie nadal możemy korzystać z metod, które nie są oznaczone jako prywatne.

```
1  pomerania = SilnikOkretowyPrywatny()
2  # Aktualnie silniki: Silniki STOP
3
4  pomerania.silniki_naprzod()
5  # Zmiana silników
6  # Aktualnie silniki: Cała naprzód
7
8  pomerania.silniki_naprzod()
9  # Aktualnie silniki: Cała naprzód
10
11 pomerania.silniki_naprzod()
12 # Aktualnie silniki: Cała naprzód
13
14 pomerania.silniki_wstecz()
15 # Zmiana silników
16 # Aktualnie silniki: Silniki STOP
17
18 pomerania.silniki_wstecz()
19 # Zmiana silników
20 # # Aktualnie silniki: Cała wstecz
21
22 pomerania.silniki_wstecz()
23 # Aktualnie siliniki: Cała wstecz
```

Gdy spróbujemy użyć metod bądź pól ustawionych jako prywatne, okaże się, że wszystko działa bezproblemowo. Ostrzeżenie wysyłane przez środowisko to jedyny sygnał informujący nas, że coś jest nie w porządku.

```
pomerania._stan_silnika = "Silnik zniszczony"
print(pomerania._stan_silnika)
# Silnik zniszczony
```

Jeśli wykorzystamy zmienną lub metodę rozpoczynającą się od "_" (poza klasą, w której są zadeklarowane), możemy niecelowo doprowadzić do błędu działania programu.

Polimorfizm

Polimorfizm oznacza wielopostaciowość. Pozwala on zastosować jedną jednostkę (metodę, operator lub obiekt) do reprezentowania różnych typów w różnych scenariuszach.

Przykład 2

Przeanalizujmy przypadek, w którym tworzymy klasę `Statek`, opisującą ogólnie każdy obiekt pływający, a także klasy dziedziczące `Bryg` oraz `Fregata` (bryg oraz fregata to typy okrętów). Inicjalizację obiektów wykonamy za pomocą konstruktora zdefiniowanego w klasie `Statek`, wykorzystując funkcję `super()`. Pozwala to na używanie metod z klasy `Statek` w klasie potomnej. Dodatkowo przetestujemy działanie metod `info()` i `test()`. Dla celów wizualizacji użyjemy funkcji wypisywania danych na ekranie, z zastosowaniem techniki f-string.

```
class Statek:
    def __init__(self, rok_wodowania):
        self._rok_wodowania = rok_wodowania
        print(f"Utworzono obiekt {self}.")

    def nowy_rok(self):
        self._rok_wodowania += 1

    def rok_wodowania(self):
        return self._rok_wodowania

class Bryg(Statek):
    def __init__(self, rok_wodowania):
        super().__init__(rok_wodowania)
        self._typ = "Bryg/2 maszty"

    def info(self):
        print(f"Metoda w klasie {__class__.__name__}")
        print(f"Obiekt {self} - typ: {self._typ}")
```

```
        print(f"Rok wodowania = {super().rok_wodowania()}")

    def test(self):
        print("To jest wywołanie test-Bryg (rok_wodowania + 20)
        print(f"wynik = {super().rok_wodowania()+20}")

class Fregata(Statek):
    def __init__(self, rok_wodowania):
        super().__init__(rok_wodowania)
        self._typ = "Fregata/3 maszty"

    def info(self):
        print(f"Metoda w klasie {__class__.__name__}")
        print(f"Obiekt {self} - typ: {self._typ}")
        print(f"Rok wodowania = {super().rok_wodowania()}")

    def test(self):
        print("To jest wywołanie test-Fregata (rok_wodowania +
        print(f"wynik = {super().rok_wodowania()+40}")


# definiujemy obiekty oraz tworzymy listę --- wartości ID obiek
statek01 = Fregata(2001)
statek02 = Fregata(2002)
statek03 = Bryg(2010)
statek04 = Bryg(2012)
sts = Bryg(1992)
cutty_sark = Fregata(1869)

# Utworzono obiekt <__main__.Fregata object at 0x7f22a2ffac50>.
# Utworzono obiekt <__main__.Fregata object at 0x7f22a410dba8>.
# Utworzono obiekt <__main__.Bryg object at 0x7f22a410dc18>.
# Utworzono obiekt <__main__.Bryg object at 0x7f22a2ffadd8>.
# Utworzono obiekt <__main__.Bryg object at 0x7f22a2ffada0>.
# Utworzono obiekt <__main__.Fregata object at 0x7f22a2ffad30>.

spis_statkow = [statek01, sts, statek02, statek03, cutty_sark,
```

Możemy teraz napisać kod, który sprawdzi, czy obiekty są uznawane za instancje klasy `Statek`. W języku Python sprawdzenie przynależności danego obiektu do klasy wykonuje się metodą `isinstance()`.

```
for statek in spis_statkow:
    print(f"isinstance({statek}, Bryg) = {isinstance(statek, Br
    print(f"isinstance({statek}, Fregata) = {isinstance(statek,
    print(f"isinstance({statek}, Statek) = {isinstance(statek,
    print("----------")


# isinstance(<__main__.Fregata object at 0x7f22a2ffac50>, Bryg)
# isinstance(<__main__.Fregata object at 0x7f22a2ffac50>, Frega
# isinstance(<__main__.Fregata object at 0x7f22a2ffac50>, State
# ----------
# isinstance(<__main__.Bryg object at 0x7f22a2ffada0>, Bryg) =
# isinstance(<__main__.Bryg object at 0x7f22a2ffada0>, Fregata)
# isinstance(<__main__.Bryg object at 0x7f22a2ffada0>, Statek)
# ----------
# isinstance(<__main__.Fregata object at 0x7f22a410dba8>, Bryg)
# isinstance(<__main__.Fregata object at 0x7f22a410dba8>, Frega
# isinstance(<__main__.Fregata object at 0x7f22a410dba8>, State
# ----------
# isinstance(<__main__.Bryg object at 0x7f22a410dc18>, Bryg) =
# isinstance(<__main__.Bryg object at 0x7f22a410dc18>, Fregata)
# isinstance(<__main__.Bryg object at 0x7f22a410dc18>, Statek)
# ----------
# isinstance(<__main__.Fregata object at 0x7f22a2ffad30>, Bryg)
# isinstance(<__main__.Fregata object at 0x7f22a2ffad30>, Frega
# isinstance(<__main__.Fregata object at 0x7f22a2ffad30>, State
# ----------
# isinstance(<__main__.Bryg object at 0x7f22a2ffadd8>, Bryg) =
# isinstance(<__main__.Bryg object at 0x7f22a2ffadd8>, Fregata)
# isinstance(<__main__.Bryg object at 0x7f22a2ffadd8>, Statek)
```

Możemy również pokazać polimorfizm – w tym celu, dla wszystkich obiektów z listy `spis_statkow`, dziedziczących po klasie `Statek`, wywołamy metodę `test()`. Zauważ, że metoda działa inaczej dla obiektów różnych klas.

```
for statek in spis_statkow:
    print("----------------------------")
    statek.test()
    print("-----")
    statek.info()
```

```
6
7
8  # ------------------------------
9  # To jest wywołanie test-Fregata (rok_wodowania + 40)
10 # wynik = 2041
11 # -----
12 # Metoda w klasie Fregata
13 # Obiekt <__main__.Fregata object at 0x7f22a2ffac50> - typ: Fre
14 # Rok wodowania = 2001
15 # ------------------------------
16 # To jest wywołanie test-Bryg (rok_wodowania + 20)
17 # wynik = 2012
18 # -----
19 # Metoda w klasie Bryg
20 # Obiekt <__main__.Bryg object at 0x7f22a2ffada0> - typ: Bryg/2
21 # Rok wodowania = 1992
22 # ------------------------------
23 # To jest wywołanie test-Fregata (rok_wodowania + 40)
24 # wynik = 2042
25 # -----
26 # Metoda w klasie Fregata
27 # Obiekt <__main__.Fregata object at 0x7f22a410dba8> - typ: Fre
28 # Rok wodowania = 2002
29 # ------------------------------
30 # To jest wywołanie test-Bryg (rok_wodowania + 20)
31 # wynik = 2030
32 # -----
33 # Metoda w klasie Bryg
34 # Obiekt <__main__.Bryg object at 0x7f22a410dc18> - typ: Bryg/2
35 # Rok wodowania = 2010
36 # ------------------------------
37 # To jest wywołanie test-Fregata (rok_wodowania + 40)
38 # wynik = 1909
39 # -----
40 # Metoda w klasie Fregata
41 # Obiekt <__main__.Fregata object at 0x7f22a2ffad30> - typ: Fre
42 # Rok wodowania = 1869
43 # ------------------------------
44 # To jest wywołanie test-Bryg (rok_wodowania + 20)
45 # wynik = 2032
46 # -----
47 # Metoda w klasie Bryg
```

```
# Obiekt <__main__.Bryg object at 0x7f22a2ffadd8> - typ: Bryg/2
# Rok wodowania = 2012
```

Dodatkowo możemy wykonać metodę, która jest zdefiniowana w klasie `Statek`, a wynik jej działania zobaczymy podczas uruchamiania kolejnej metody.

```
for statek in spis_statkow:
    statek.nowy_rok()
# tu nie ma żadnej informacji zwrotnej, metoda tylko aktualizuj

for statek in spis_statkow:
    statek.info()


# Metoda w klasie Fregata
# Obiekt <__main__.Fregata object at 0x7f22a2ffac50> - typ: Fre
# Rok wodowania = 2002
# Metoda w klasie Bryg
# Obiekt <__main__.Bryg object at 0x7f22a2ffada0> - typ: Bryg/2
# Rok wodowania = 1993
# Metoda w klasie Fregata
# Obiekt <__main__.Fregata object at 0x7f22a410dba8> - typ: Fre
# Rok wodowania = 2003
# Metoda w klasie Bryg
# Obiekt <__main__.Bryg object at 0x7f22a410dc18> - typ: Bryg/2
# Rok wodowania = 2011
# Metoda w klasie Fregata
# Obiekt <__main__.Fregata object at 0x7f22a2ffad30> - typ: Fre
# Rok wodowania = 1870
# Metoda w klasie Bryg
# Obiekt <__main__.Bryg object at 0x7f22a2ffadd8> - typ: Bryg/2
# Rok wodowania = 2013
```

Dla zainteresowanych

W języku Python przykładem polimorfizmu jest operator „+”. Działa on w różny sposób w zależności od typu danych.

```
# przykład działania operatora + dla typów numerycznych
print(3 + 3)
```

```
# 6
print(3 + 3.0)
# 6.0
# przykład działania operatora + dla typów znakowych
print("3" + "3")
# 33
```

Dla zainteresowanych

Innym przykładem polimorfizmu w języku Python jest funkcja `len()`, która działa dla różnych typów danych w ten sam sposób.

```
# przykład działania
# str
print(len("Python to język programowania."))
# 30
# dict
print(len({1: "Python", 2: "Linux", 3: "Open Source"}))
# 3
# list
print(len([1, 3, 4, [1, 1], "Python", True, 3.14]))
# 7
# tuple
print(len((1, 3, 4, "Python", True, 3.14)))
# 6
# niektóre typy danych nie są obsługiwane przez funkcję len()
# int
print(len(4))
# Traceback (most recent call last):
#   File "d:\Programming\VSCProjects\Poprawianie\0431\test.py",
#     print(len(4))
# TypeError: object of type 'int' has no len()
```

Już wiesz

Podsumujmy najważniejsze elementy tego e-materiału:

- język programowania Python jest w pełni językiem obiektowym,
- zawiera narzędzia i sposoby na bezproblemowe stosowanie obiektowego paradygmatu programowania,
- pozwala wygodnie tworzyć klasy, obiekty, właściwości i metody.

Słownik

f-string

sposób zapisywania zmiennych w łańcuchu znaków przeznaczonym do wyświetlenia za pomocą funkcji `print()`; dostępny od wersji 3.6 języka Python; polecenie wykorzystujące mechanizm `f-string` ma postać: `f"Napis, a w nim {zmienna} do wypisania"`; dokładnie opisany w dokumencie PEP 498 – *Literal String Interpolation*

Prezentacja multimedialna

Polecenie 1

Zapoznaj się z prezentacją. Przeanalizuj zachowanie zasad poliformizmu na przykładzie metody `maszty_opis()` i hermetyzacji na przykładzie pól `_maszty` i `_typ`.

1

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Użyjemy definicji klasy `Statek`, która w konstruktorze przyjmuje i ustawia wartości pól `_typ` i `_maszty`. Jeśli typ nie zawiera się w zakresie $\langle 0,\ 3 \rangle$ to jest ustawiany na 4, co oznacza, że jest to nieznany typ statku. Klasa posiada także metody `typ_opis()`, `info()` i `maszty_opis()`, które są odpowiedzialne za wyświetlanie informacji o statku.

```
1  class Statek:
2      def __init__(self,
   maszty=0, typ=0):
3          if type(typ) is
   not int or not 0 <= typ <=
   3:
4              typ = 4
5          self._typ = typ
6          self._maszty =
   maszty
7          print(f"Utworzono
   obiekt (statek) o ID:
   {id(self)}")
8
9      def typ_opis(self):
10         opisy = { 0:
   "Tylko kadłub",
11                   1:
   "Bryg",
```

```
                    2:
"Klipper",
                    3:
"Korweta",
                    4:
"Nieznany",
                    }
        return
opisy[self._typ]

    def info(self):
        print(f"Informacje
o obiekcie klasy
{__class__.__name__} (ID:
{id(self)})")
        print(f"Typ:
{self.typ_opis()}")

    def maszty_opis(self):
        if
type(self._maszty) is int:

print(f"Informacja o
liczbie masztów:
{self._maszty}")
        else:
            print("Nie
znamy liczby masztów.")
```

2

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Tworzymy obiekt `s1` klasy `Statek`. Obiekt ten ma 0 masztów i reprezentuje typ `Tylko kadłub`:

```
s1 = Statek()
```

Jest tak, ponieważ przyjmuje wartości domyślne, gdyż nie ustawiliśmy żadnych

wartości pól dla tego obiektu klasy `Statek`.

3

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Tworzymy obiekt s2 klasy `Statek` mający 2 maszty i będący typu `Bryg`:

```
1 s2 = Statek(2, 1)
```

4

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Tworzymy obiekt s3 klasy `Statek` mający 0 masztów i będący typu `Nieznany` (ponieważ 5 jest większe niż 3):

```
1 s3 = Statek(typ=5)
```

5

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Tworzymy obiekt s4 klasy `Statek`, którego wartość pola `_maszty` wynosi `Brak`, a jego typ to `Klipper`:

```
1 s4 = Statek("Brak", 2)
```

Zwróć uwagę, że typ nie jest wyrażony liczbą.

6

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Sprawdzamy informacje dostępne dla obiektów:

```
s1.info()
s2.info()
s3.info()
s4.info()
```

7

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

Sprawdzamy działanie metody `maszty_opis()` dla obiektów:

```
s1.maszty_opis()
s2.maszty_opis
s3.maszty_opis()
s4.maszty_opis()
```

8

Materiał audio dostępny pod adresem:

https://zpe.gov.pl/b/PJGzxgfR4

W przedstawionym kodzie zastosowaliśmy hermetyzację i polimorfizm, które są podstawowymi założeniami obiektowego paradygmatu programowania. Cały kod programu:

```
class Statek:
    def __init__(self, maszty=0, typ=0):
```

```
        if type(typ) is not int or not 0 <= typ <= 3:
            typ = 4
        self._typ = typ
        self._maszty = maszty
        print(f"Utworzono obiekt (statek) o ID: {id(self)}")

    def typ_opis(self):
        opisy = { 0: "Tylko kadłub",
                  1: "Bryg",
                  2: "Klipper",
                  3: "Korweta",
                  4: "Nieznany",
                  }
        return opisy[self._typ]

    def info(self):
        print(f"Informacje o obiekcie klasy {__class__.__name__} (ID: {id(self)})")
        print(f"Typ: {self.typ_opis()}")

    def maszty_opis(self):
        if type(self._maszty) is int:

print(f"Informacja o liczbie masztów: {self._maszty}")
        else:
            print("Nie znamy liczby masztów.")

```

```
s1 = Statek()
s2 = Statek(2, 1)
s3 = Statek(typ=5)
s4 = Statek("Brak", 2)

s1.info()
s2.info()
s3.info()
s4.info()
s1.maszty_opis()
s2.maszty_opis()
s3.maszty_opis()
s4.maszty_opis()
```


Polecenie 2

Przygotuj notatkę, w której podsumujesz najważniejsze informacje zawarte w prezentacji

Sprawdź się

Pokaż ćwiczenia:

Ćwiczenie 1

Przeanalizuj przedstawiony kod.

```
class Counter:
    def __init__(self, count):
        # miejsce na definicje pola instancji klasy

    def t(self):
        print(f"Wartość  => {self."tu wprowadź nazwę zdefiniowa


# wykonanie metody
c = Counter(5)
c.t()
```

Wskaż, która definicja pola instancji klasy jest poprawna, gdy chcemy oznaczyć ją jako zmienną, która nie powinna być modyfikowana poza klasą.

- ◯ `self.__count = count`
- ◯ `self.count = count`
- ◯ `self._count = count`

Ćwiczenie 2

Zdefiniuj dwie klasy: pierwszą o nazwie `Punkt` oraz dziedziczącą z niej klasę `KolorowyPunkt`.

Niech w klasie istnieje konstruktor przypisujący wartość do pól o nazwie `x` i `y` oraz metoda o nazwie `wypisz_wspolrzedne()`, wypisująca `f-string: "Położenie punktu to ({x}, {y})"`.

Niech w klasie `KolorowyPunkt` istnieje korzystający z konstruktora klasy `Punkt` (super), który dodatkowo ustawi wartość pola `kolor`. Jego zadaniem jest sprawdzenie, czy wartość jest typu `string`. W przeciwnym wypadku powinien przypisać domyślną wartość `"czerwony"` do pola `kolor`. Dla klasy zdefiniuj również metodę o nazwie `wypisz_kolor()`, wypisująca `f-string: "Kolor punktu to {kolor}"`.

Specyfikacja problemu:

Dane:

Pole instancji klasy `A`:

- `x` – liczba całkowita
- `y` – liczba całkowita

Pole instancji klasy `B`:

- `x` – liczba całkowita
- `y` – liczba całkowita
- `kolor` – łańcuch znaków

Wynik:

Program, na standardowe wyjście, wypisuje wartości przez wywołanie metody `wypisz_wspolrzedne()` dla obiektu klasy `Punkt` i metod `wypisz_wspolrzedne()`, `wypisz_kolor()` dla obiektów klasy `KolorowyPunkt`.

Przetestuj program dla obiektu klasy `Punkt` z wartością pola x równą 2 i pola y równą 1, obiektu klasy `KolorowyPunkt` z wartością pól: x = 3, y = 7, `kolor` = -5 i obiektu klasy B z wartością pól: x = 8, y = -3, `kolor` = „niebieski”.

Twoje zadania

1. Program wypisuje wartości zwrócone przez wywołanie metod `wypisz_wspolrzedne()` i `wypisz_kolor()` klas `Punkt` i `KolorowyPunkt`.

```
class Punkt:
    pass


class KolorowyPunkt(Punkt):
    pass


punkt1 = Punkt(2, 1)
punkt2 = KolorowyPunkt(3, 7, -5)
punkt3 = KolorowyPunkt(8, -3, "niebieski")

punkt1.wypisz_wspolrzedne()
punkt2.wypisz_wspolrzedne()
```

```
```

Ćwiczenie 3

Korzystając z przedstawionego kodu, utwórz klasę `Policjant`, która będzie dziedziczyć po klasie `Pracownik`. W klasie `Policjant` zadeklaruj konstruktor, który będzie przyjmował dwa parametry typu `String`: `imie` oraz `nazwisko`. Zadbaj, aby pole `miejscePracy` w klasie `Policjant` było ustawione na `komisariat`. Następnie, przez nadpisanie metody `przedstawSię()`, popraw błąd językowy tak, aby program wypisywał komunikat `Nazywam się {imie} {nazwisko}. Pracuję na komisariacie.` Nie modyfikuj istniejącej klasy `Pracownik` oraz funkcji głównej programu. Program przetestuj dla imienia Jan i nazwiska Kowalski.

Specyfikacja problemu:

Dane:

- `imie` – imię; ciąg znaków
- `nazwisko` – nazwisko; ciąg znaków

Wynik:

Program, na standardowe wyjście, wypisuje zadany komunikat.

Twoje zadania

1. Program wypisuje komunikat z imieniem i nazwiskiem policjanta, a także jego miejsce pracy

```
class Pracownik:
    def __init__(self, imie: str, nazwisko: str, miejsce_pracy: str):
        self.imie = imie
        self.nazwisko = nazwisko
        self.miejsce_pracy = miejsce_pracy

    def przedstaw_sie(self):
        print(f"Nazywam się {self.imie} {self.nazwisko}. Pracuję w {self.miejsce_pracy}")
```

```
9
10 class Policjant(Pracownik):
11     pass # Tu wpisz swój kod
12
```

```
1
```

Dla nauczyciela

Autor: Adam Jurkiewicz
Przedmiot: Informatyka

Temat: Zasady programowania obiektowego w języku Python

Grupa docelowa:

Szkoła ponadpodstawowa, liceum ogólnokształcące, technikum, zakres rozszerzony

Podstawa programowa:

Cele kształcenia – wymagania ogólne

> II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera oraz innych urządzeń cyfrowych: układanie i programowanie algorytmów, organizowanie, wyszukiwanie i udostępnianie informacji, posługiwanie się aplikacjami komputerowymi.

Treści nauczania – wymagania szczegółowe

> II. Programowanie i rozwiązywanie problemów z wykorzystaniem komputera i innych urządzeń cyfrowych.
>
> > Zakres rozszerzony. Uczeń spełnia wymagania określone dla zakresu podstawowego, a ponadto:
> >
> > > 2) stosuje zasady programowania strukturalnego i obiektowego w rozwiązywaniu problemów;

Kształtowane kompetencje kluczowe:

- kompetencje cyfrowe;
- kompetencje osobiste, społeczne i w zakresie umiejętności uczenia się;
- kompetencje matematyczne oraz kompetencje w zakresie nauk przyrodniczych, technologii i inżynierii.

Cele operacyjne (językiem ucznia):

- Prześledzisz zaawansowane zasady programowania obiektowego w języku Python.
- Zaimplementujesz definicje klas wykorzystujących techniki programowania obiektowego.
- Stworzysz program wykorzystujący techniki programowania obiektowego i definicje klas.

Strategie nauczania:

- konstruktywizm;
- konektywizm.

Metody i techniki nauczania:

- dyskusja;
- rozmowa nauczająca z wykorzystaniem multimedium i ćwiczeń interaktywnych;
- ćwiczenia praktyczne.

Formy pracy:

- praca indywidualna;
- praca w parach;
- praca w grupach;
- praca całego zespołu klasowego.

Środki dydaktyczne:

- komputery z głośnikami, słuchawkami i dostępem do internetu;
- zasoby multimedialne zawarte w e-materiale;
- tablica interaktywna/tablica, pisak/kreda;
- oprogramowanie dla języka Python 3 (lub nowszej wersji), w tym PyCharm lub IDLE.

Przebieg lekcji

Przed lekcją:

1. **Przygotowanie do zajęć.** Nauczyciel loguje się na platformie i udostępnia e-materiał: „Zasady programowania obiektowego w języku Python". Uczniowie zapoznają się z treściami w sekcji „Przeczytaj" w kontekście programowania.

Faza wstępna:

1. Nauczyciel wyświetla i odczytuje temat lekcji oraz cele zajęć. Prosi uczniów o sformułowanie kryteriów sukcesu.
2. Prowadzący prosi uczniów, aby zgłaszali swoje propozycje pytań do tematu. Jedna osoba może zapisywać je na tablicy. Gdy uczniowie wyczerpią swoje pomysły, a pozostały jakieś ważne kwestie do poruszenia, nauczyciel je dopowiada.

Faza realizacyjna:

1. Nauczyciel wyświetla zawartość sekcji „Przeczytaj". Na forum klasy uczniowie analizują przedstawione w niej rozwiązania Przykładów 1 i 2. A następnie testują je na swoich komputerach.

2. **Praca z multimedium.** Nauczyciel wyświetla zawartość sekcji „Prezenacja multimedialna”. Uczniowie zapoznają się indywidualnie z treścią prezentacji, analizują zachowanie zasad polimorfizmu na przykładzie metody `maszty_opis()` i hermetyzacji na przykładzie pól `_maszty` i `_typ`. Nauczyciel wyjaśnia ewentualne niezrozumiałe kwestie.
3. **Ćwiczenie umiejętności.** Prowadzący zapowiada uczniom, że będą rozwiązywać ćwiczenie nr 1 z sekcji „Sprawdź się”. Każdy z uczniów robi to samodzielnie. Po ustalonym czasie następuje porównanie napisanych kodów podczas wspólnego omówienia rozwiązań.
4. Uczniowie w parach wykonują ćwiczenia nr 2. Nauczyciel sprawdza poprawność wykonanych zadań, omawiając je wraz z uczniami.

Faza podsumowująca:

1. Wybrany uczeń podsumowuje zajęcia, zwracając uwagę na nabyte umiejętności, omawia ewentualne problemy podczas rozwiązania ćwiczeń z programowania w języku Python.

Praca domowa:

1. Uczniowie wykonują ćwiczenie nr 3 z sekcji „Sprawdź się”.

Materiały pomocnicze:

- Oficjalna dokumentacja techniczna dla języka Python 3 (lub nowszej wersji).
- Oficjalna dokumentacja techniczna dla oprogramowania PyCharm lub IDLE.

Wskazówki metodyczne:

- Treści w sekcji „Przeczytaj” można wykorzystać jako podsumowanie i utrwalenie wiedzy uczniów.