KURS PROGRAMOWANIA W JĘZYKU PYTHON

TYDZIEŃ 6 – WSTĘP DO PROGRAMOWANIA OBIEKTOWEGO

1. Konstrukt if __name__ == '__main__'

If __name__ == '__main__' to konstrukcja w języku Python, która pozwala na wykonywanie kodu tylko wtedy, gdy skrypt jest uruchamiany bezpośrednio, a nie importowany jako moduł do innego skryptu. Jest to przydatne, gdy chcemy, aby pewne fragmenty kodu wykonywały się tylko wtedy, gdy plik jest uruchamiany jako główny program, a nie jako moduł importowany przez inny skrypt. Jej użycie oraz funkcję najlepiej wytłumaczyć na przykładach.

Przykład 1:

```
def operation1():
    print("Operation 1")

def operation2():
    print("Operation 2")

if __name__ == '__main__':
    operation1()
    operation2()
```

W tym przypadku, gdy skrypt jest uruchamiany bezpośrednio, zostaną wykonane obie operacje print. Jednak, jeśli ten skrypt zostanie zaimportowany do innego skryptu, tylko definicje funkcji będą dostępne, a same operacje nie zostaną wykonane automatycznie (czyli nie dostaniemy printów).

Przykład 2:

```python
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

if __name__ == '__main__':
    # Testy funkcji
    print("Testing add function:")
    print("2 + 3 =", add(2, 3))
    print("5 + 7 =", add(5, 7))

    print("\nTesting subtract function:")
    print("8 - 4 =", subtract(8, 4))
    print("10 - 6 =", subtract(10, 6))
```

W tym przykładzie mamy moduł z dwiema funkcjami: **add()** i **subtract()**. Gdy skrypt jest uruchamiany bezpośrednio, wykonuje się blok testowy, który sprawdza poprawność działania tych funkcji. Jeśli moduł jest zaimportowany, testy nie są automatycznie wykonywane. Technika ta jest przydatna do testowania funkcji podczas ich rozwijania i debugowania.

Klasy i obiekty to fundamentalne pojęcia w programowaniu obiektowym, w tym także w języku Python.

2. Klasy i obiekty

Klasa

Klasa to szkielet, który definiuje **cechy i zachowanie obiektów.** Opisuje, jakie cechy (parametry) i jakie operacje (metody) mogą być wykonywane na tych obiektach. Z klasami związanych jest kilka dodatkowych pojęć:

- **Atrybuty**, albo parametry, klasy definiują zmienne związane z danym obiektem.
- **Metody** klasy to funkcje związane z danym obiektem.
- **Konstruktory** klasy to specjalne metody konstruktora np. **__init__()**, które są wywoływane podczas tworzenia nowego obiektu danej klasy.

Ważną częścią klas jest parametr **self,** który omówimy w kolejnej sekcji przy omawianiu metody **__init__**. Przykład zobaczysz jednak wcześniej podczas omawiania obiektów.

Do definiowania klas używamy słowa kluczowego **"class"**.

Obiekt

Obiekt jest instancją konkretnej klasy. Każdy obiekt ma swoje własne wartości atrybutów i może wywoływać metody zdefiniowane w klasie. Możemy myśleć o obiekcie jak o przedmiocie lub bycie, który ma swoje własne **cechy** (atrybuty) i **umiejętności** (metody). Na przykład, może to być samochód. Samochód może mieć atrybuty takie jak kolor, marka i prędkość. Może też mieć umiejętności takie jak przyspieszanie czy hamowanie.

Kiedy mówimy o klasach, mówimy właśnie o planie dla tych obiektów. Jest to jak instrukcja do zbudowania samochodu. Klasa określa, jakie atrybuty i metody będą miały obiekty z niej tworzone. Na przykład, możemy stworzyć **klasę** o nazwie **Car**, która będzie zawierała atrybuty takie jak color (kolor), brand (marka) i metody takie jak accelerate (przyspiesz) i brake (hamuj).

Kiedy już mamy zdefiniowaną klasę, możemy tworzyć konkretne obiekty z niej, takie jak **my_car = Car()**. Ten obiekt będzie miał wszystkie atrybuty i umiejętności zdefiniowane w klasie Car.

Przykład:

Zdefiniujmy klasę Car.

```
class Car:
    def __init__(self, color, brand, speed=0):
        self.color = color
        self.brand = brand
        self.speed = speed

    def accelerate(self, increment):
        self.speed += increment

    def brake(self, decrement):
        if self.speed - decrement >= 0:
            self.speed -= decrement
        else:
            print("The car cannot brake further.")
```

Klasa **Car** ma 3 atrybuty color (kolor), brand (marka) i speed (prędkość) oraz 2 metody: accelerate (przyspiesz) i brake (hamuj), które zmieniają prędkość samochodu.

Stwórzmy teraz 2 obiekty: **my_car** i **your_car.**

```
# Tworzenie obiektów (instancji) klasy Car
my_car = Car("red")
your_car = Car("blue", 50)

# Wyświetlenie informacji o samochodach bez atrybutu "brand"
print("My car - Color:", my_car.color, ", Speed:", my_car.speed)
print("Your car - Color:", your_car.color, ", Speed:", your_car.speed)

# Przyspieszanie i hamowanie samochodów
my_car.accelerate(20)
your_car.brake(10)

# Wyświetlenie zaktualizowanych informacji o samochodach
print("After acceleration and braking:")
print("My car - Speed:", my_car.speed)
print("Your car - Speed:", your_car.speed)
```

Tworzymy dwie instancje klasy Car: **my_car** i **your_car**, z różnymi wartościami atrybutów. Następnie przyspieszamy mój samochód o 20 jednostek prędkości i hamujemy Twój samochód o 10 jednostek prędkości. Na koniec wyświetlamy zaktualizowane informacje o prędkości samochodów.

3. Metody klas: __init__, __str__

Specjalne metody klas, takie jak **__init__()** i **__str__()**, są często używane w języku Python do dostosowywania zachowania klas do różnych potrzeb.

__init__

Metoda **__init__()** to specjalna metoda, która jest wywoływana automatycznie podczas tworzenia nowego obiektu danej klasy. Jest to tzw. **konstruktor klasy.** metoda ta jest jak magiczne urządzenie, które inicjuje wszystkie rzeczy w nowym obiekcie. Kiedy tworzysz nowy obiekt w klasie, ta metoda jest wywoływana automatycznie, pomagając ustawiać początkowe wartości dla obiektu.

Przykład metody __init__:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Tworzenie nowego obiektu
person1 = Person("Alice", 30)
print(person1.name)  # Output: Alice
print(person1.age)   # Output: 30
```

W powyższym przykładzie, metoda **__init__** przyjmuje dwa argumenty: name i age, które są przypisywane do atrybutów name i age **obiektu Person**.

__str__

Metoda **__str__** jest funkcją, która mówi Pythonowi, jak opisać obiekt w sposób czytelny. Kiedy chcesz wyświetlić obiekt w czytelny sposób, metoda ta jest wywoływana automatycznie.

Przykład metody __str__:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Person: {self.name}, Age: {self.age}"

# Tworzenie nowego obiektu
person1 = Person("Bob", 25)
print(person1)  # Output: Person: Bob, Age: 25
```

W tym przykładzie, gdy drukujemy obiekt person1, metoda __str__ jest automatycznie wywoływana, zwracając czytelny opis obiektu.

4. Definiowanie metod klas

Metody klas to funkcje zdefiniowane wewnątrz klasy, które definiują zachowanie obiektów danej klasy. Te metody mogą manipulować atrybutami obiektu, wykonując różne operacje.

Metody klas definiują zachowanie obiektów danej klasy, umożliwiając manipulację ich stanem oraz wykonywanie różnych operacji z nimi związanych. Są one kluczowym elementem programowania obiektowego w Pythonie.

Przykład:

Aby zdefiniować własną metodę w klasie, musimy najpierw zdefiniować klasę słowem kluczowym **class**, a następnie zdefiniować funkcje w tej klasie używając wcięcia i słowa kluczowego **def.**

```
class NazwaKlasy:
    def nazwa_metody(self, parametry):
        # ciało metody
```

class NazwaKlasy: rozpoczynamy definicję klasy używając słowa kluczowego **class**, a następnie podajemy nazwę klasy.

def nazwa_metody(self, parametry): definiujemy metodę używając słowa kluczowego def, podając nazwę metody oraz ewentualne parametry, które metoda będzie przyjmować. Parametr self jest specjalnym parametrem, który oznacza instancję klasy i jest wymagany we wszystkich metodach, aby odwoływać się do atrybutów i innych metod obiektu.

 # ciało metody: tutaj umieszczamy kod, który będzie wykonywany, gdy metoda zostanie wywołana.

5. Assert i Unittest

Instrukcja assert

Assert to instrukcja w języku Python, która służy do sprawdzania warunków. Jeśli warunek jest fałszywy, assert podnosi wyjątek AssertionError w celu wskazania na niezgodność w kodzie. Jest to przydatne narzędzie do automatycznego sprawdzania poprawności kodu w trakcie jego wykonywania.

Assert używane jest jest używane do sprawdzania warunków i zapewnienia, że spełniają one nasze oczekiwania w trakcie działania programu, jest to przydatne narzędzie do debugowania oraz do testów jednostkowych z użyciem bibliotek np. **unittest.** Biblioteka unittest zostanie omówiona poniżej.

Przykład:

W tym przykładzie użyjemy assert do sprawdzienia, czy nie jest wykonywane dzielenie przez 0.

```
def divide(a, b):
    assert b != 0, "Error: division by zero"
    return a / b

result = divide(10, 2)
print("Result:", result)  # Output: Result: 5.0

result = divide(10, 0)  # AssertionError: Error: division by zero
```

W tym przykładzie, **assert b != 0** sprawdza, czy dzielnik b nie jest zerem. Jeśli jest zerem, podnosi wyjątek **AssertionError,** informując o błędzie dzielenia przez zero.

Unittest

Testy jednostkowe są narzędziem programistycznym, które pomaga w weryfikacji poprawności poszczególnych fragmentów kodu, tj. jednostek, takich jak funkcje czy klasy. Unittest to moduł wbudowany w język Python, który umożliwia pisanie i uruchamianie testów jednostkowych.