Notatki do egzaminu z **Programowania Obiektowego** w **Pythonie**

(wersja rozbudowana, z przykładami i wyjaśnieniami „po ludzku")

> **WAŻNE**: Poniższe notatki zawierają szczegółowe omówienie najważniejszych zagadnień związanych z programowaniem obiektowym w Pythonie. Zwracaj uwagę na sekcje wyróżnione jako **WAŻNE** lub **KLUCZOWE**, ponieważ tam znajdują się kluczowe informacje przydatne na egzaminie.

---

1. Obiekty w Pythonie

W Pythonie wszystko jest obiektem – zarówno liczby, ciągi znaków (stringi), funkcje, jak i klasy. Każdy obiekt posiada cztery główne cechy:

1. **Tożsamość (identity)**
   - To unikalny identyfikator obiektu w pamięci.
   - Porównujemy za pomocą operatora `is`, np. `x is y`.
   - Jeśli `x is y` zwraca `True`, to znaczy, że obie zmienne wskazują na dokładnie ten sam obiekt (ten sam obszar pamięci).
2. **Typ (type)**
   - Typ obiektu determinuje, co można z nim zrobić (jakie ma dostępne metody, jak może być użyty).
   - Niektóre typy są **mutowalne (mutable)**, czyli można zmieniać ich stan (np. listy), a inne **niemutowalne (immutable)**, np. liczby, stringi, krotki.
3. **Stan (value)**
   - To wartość przechowywana przez obiekt (np. liczba 5, ciąg znaków `"tekst"`, lista `[1, 2, 3]` itd.).
4. **Zachowanie (behavior)**
   - Metody i operacje, jakie można wykonywać na obiekcie (np. dla list: `append`, `sort`, `pop`).

> **Ciekawostka/WAŻNE**:
> Python stosuje **internalizację** dla małych liczb całkowitych oraz literałów łańcuchowych, co oznacza, że przechowuje je w pamięci tylko raz i współdzieli je między różnymi zmiennymi (jeżeli wartości są identyczne). Ułatwia to zarządzanie pamięcią.

Przykład sprawdzania tożsamości

```
x = "tekst"
y = "tekst"
```

```
print(x == y)   # True – wartości (stany) są takie same
print(x is y)   # True lub False – zależy od internalizacji
```

---

2. Zmienne w Pythonie

W Pythonie **zmienne nie mają typu** – to **obiekty** mają typy. Zmienna jest jedynie **etykietą** (referencją), która wskazuje na dany obiekt w pamięci. Kilka kluczowych informacji:

- Zmienna zawsze wskazuje na jakiś obiekt (nawet jeśli tym obiektem jest `None`).
- Możemy dowolnie zmieniać obiekt, na który wskazuje dana zmienna (dlatego Python nazywany jest językiem **dynamicznym**).
- Przypisanie `a = b` oznacza: „zmienna `a` wskazuje na ten sam obiekt co `b`".

> **WAŻNE**:
> Jeżeli obiekt jest mutowalny (np. lista) i dokonamy zmiany przez jedną zmienną, zobaczymy tę zmianę również przez drugą zmienną, która wskazuje na ten sam obiekt.

Przykład przypisania i wspólnego obiektu

```
a = 18      # 'a' wskazuje na obiekt 18
p = 7.5     # 'p' wskazuje na obiekt 7.5
q = p       # 'q' wskazuje teraz na ten sam obiekt co 'p'

print(q is p)  # True, bo obie zmienne wskazują na ten sam obiekt
```

---

3. Typy danych w Pythonie

Python jest językiem **dynamicznie typowanym** i **interpretowanym**:

1. **Dynamiczne typowanie**
    - Oznacza, że kontrola typów (czy coś jest liczbą, stringiem itp.) odbywa się w **czasie wykonywania** programu (ang. *runtime*).
    - Dzięki temu można szybciej pisać kod, ale trzeba uważać na błędy wynikające z nieoczekiwanych typów.
2. **Interpretowany**
    - Kod Pythona jest wykonywany linijka po linijce przez interpreter, a nie kompilowany do kodu maszynowego przed uruchomieniem.

Przykładowy błąd typów

```
3 + '4'  # TypeError: nie można dodać liczby całkowitej i łańcucha znaków
```

> **WAŻNE**:
> W Pythonie każda próba wykonania operacji między obiektami niezgodnych typów kończy się błędem (`TypeError`), chyba że dany operator został przeciążony (o tym w rozdziale **Przeciążanie operatorów**).

Przykład typów danych i jak z nich korzystać

- **Liczby całkowite (int)**: `x = 42`
- **Liczby zmiennoprzecinkowe (float)**: `x = 3.14`
- **Ciągi znaków (str)**: `x = "Hello World!"`
- **Krotki (tuple)**:

```
# krotka jest niemutowalna (immutable)
moja_krotka = (1, 2, 3, "tekst")
```

- **Listy (list)**:

```
# listy są mutowalne (mutable)
moja_lista = [1, 2, 3]
moja_lista.append(4)
```

- **Słowniki (dict)**:

```
# słowniki przechowują pary klucz-wartość
moj_slownik = {
    "imie": "Jan",
    "wiek": 30
}
```

- **Zbiory (set)**:

```
# zbiory nie przechowują duplikatów
moj_zbior = {1, 2, 3, 3, 2, 1}
print(moj_zbior)  # {1, 2, 3}
```

---

4. Programowanie obiektowe (OOP)

W **Programowaniu Obiektowym** podstawowym elementem jest **obiekt**, czyli „egzemplarz (instancja) klasy”. Obiekty łączą w sobie:

- **Atrybuty (attributes)** – odpowiadają za stan (np. pola/zmienne w innych językach).
- **Metody (methods)** – funkcje zdefiniowane w obrębie klasy, które opisują zachowanie obiektu.

Tworzenie klasy w Pythonie

```
class KlasaPrzyklad:
    def __init__(self, atrybut):
        # Konstruktor klasy wywoływany przy tworzeniu obiektu
        self.atrybut = atrybut   # przypisujemy wartość do atrybutu obiektu

# Tworzymy obiekt (instancję)
obiekt = KlasaPrzyklad("wartosc")
print(obiekt.atrybut)  # "wartosc"
```

> **WAŻNE**:
> W Pythonie każda metoda w klasie ma jako pierwszy parametr `self`, który odnosi się do konkretnego obiektu (instancji), na którym dana metoda jest wywoływana. To jest **kluczowa różnica** w porównaniu do niektórych innych języków, gdzie słowo kluczowe `this` występuje „za kulisami”.

---

5. Atrybuty i metody klasy

Atrybuty klasy vs. atrybuty instancji

- **Atrybuty instancji** (np. `self.atrybut`) są unikalne dla każdego obiektu.
- **Atrybuty klasy** (zdefiniowane bezpośrednio w ciele klasy, poza metodami) są współdzielone przez wszystkie obiekty danej klasy.

```
class Przyklad:
    wspolny_atrybut = "To jest atrybut klasy"  # atrybut klasy

    def __init__(self, atrybut):
        self.atrybut = atrybut  # atrybut instancji

ob1 = Przyklad("A")
ob2 = Przyklad("B")
print(ob1.wspolny_atrybut, ob2.wspolny_atrybut)
# Oba obiekty zobaczą: "To jest atrybut klasy"
```

Metody statyczne i dekoratory

- **Metody statyczne** tworzymy za pomocą dekoratora `@staticmethod`.
  Nie przyjmują parametru `self`, ponieważ nie operują na instancji klasy.

```
class Matematyka:
    @staticmethod
    def dodaj(a, b):
        return a + b

print(Matematyka.dodaj(3, 4))  # 7
```

- **@property** – służy do tworzenia właściwości (properties), które wyglądają jak atrybuty, ale są obsługiwane przez metody.
  Dzięki temu możemy np. dodać logikę walidacji wartości przy zapisie.

```
class Osoba:
    def __init__(self, imie):
        self._imie = imie

    @property
    def imie(self):
        """Właściwość tylko do odczytu"""
        return self._imie

os = Osoba("Ala")
print(os.imie)  # "Ala"
# os.imie = "Ola"  # TypeError, bo nie ma settera
```

> **WAŻNE**:
> Dekoratory takie jak `@property`, `@staticmethod`, `@classmethod` pozwalają zmieniać lub rozszerzać zachowanie metod w klasie.

---

6. Hermetyzacja (Encapsulation)

Hermetyzacja polega na ukrywaniu szczegółów implementacji wewnątrz klasy i udostępnianiu jedynie interfejsu (metod, atrybutów „publicznych”).
W Pythonie **nie ma** ścisłych modyfikatorów dostępu (`public`, `private`, `protected`) jak w C++ czy Javie, lecz istnieją **konwencje** nazewnicze:

- `self._atrybut` – oznacza, że atrybut jest „chroniony” i nie powinno się go używać na zewnątrz klasy (ale technicznie można).
- `self.__atrybut` – Python stosuje **name mangling** (przekształcenie nazwy), by utrudnić dostęp do atrybutu w klasach dziedziczących.

```
class PrzykładHermetyzacji:
    def __init__(self):
        self.publiczny = "Dostępne wszędzie"
        self._chroniony = "Zalecane do użytku wewnętrznego"
        self.__prywatny = "Trudny dostęp z zewnątrz"

ob = PrzykładHermetyzacji()
print(ob.publiczny)
print(ob._chroniony)
# print(ob.__prywatny)  # AttributeError
```

> **WAŻNE**:
> Mimo że można obejść `__prywatny` (np. `ob._PrzykładHermetyzacji__prywatny`), to **nie jest to zalecane** i uważa się za złamanie kapsułkowania.

---

7. Dziedziczenie

Dziedziczenie pozwala tworzyć nowe, bardziej wyspecjalizowane klasy (**klasy pochodne**) na bazie istniejących klas (**klas bazowych**). Dzięki temu:

1. **Wykorzystujesz ponownie kod** z klasy bazowej.
2. **Rozszerzasz lub nadpisujesz** (ang. *override*) zachowania (metody) z klasy bazowej.

Przykład dziedziczenia

```
class Bazowa:
    def metoda(self):
        print("Metoda klasy bazowej")

class Pochodna(Bazowa):
    def metoda(self):
        # super() pozwala wywołać metodę z klasy bazowej
        super().metoda()
        print("Metoda klasy pochodnej")

p = Pochodna()
p.metoda()
# Wynik:
# Metoda klasy bazowej
# Metoda klasy pochodnej
```

> **KLUCZOWE**:
>
> - Słowo kluczowe `super()` odwołuje się do klasy bazowej i pozwala wywołać jej metody lub konstruktory.
> - Możesz tworzyć wielopoziomową hierarchię dziedziczenia.

---

8. Polimorfizm

Polimorfizm (z gr. *wiele form*) oznacza w praktyce możliwość używania różnych typów obiektów, które mają **tę samą metodę**, ale różną implementację. Dzięki temu możemy pisać bardziej uniwersalny kod.

Przykład polimorficzny

```
class Pies:
    def daj_glos(self):
        print("Hau! Hau!")
```

```
class Kot:
    def daj_glos(self):
        print("Miau!")

def zwierze_daj_glos(zwierze):
    zwierze.daj_glos()

pies = Pies()
kot = Kot()

zwierze_daj_glos(pies)  # Hau! Hau!
zwierze_daj_glos(kot)   # Miau!
```

> **KLUCZOWE**:
> W Pythonie polimorfizm jest naturalnie powiązany z **kaczym typowaniem** („jeśli coś kwacze jak kaczka i wygląda jak kaczka, to traktuj to jak kaczkę"). Ważne jest, aby obiekt miał konkretną metodę (np. `daj_glos`), a jego faktyczny typ może być dowolny.

---

9. Obsługa wyjątków

Python posiada rozbudowany mechanizm obsługi wyjątków, co pozwala na bezpieczne i czytelne zarządzanie błędami:

- `try...except` – przechwytywanie wyjątków.
- `finally` – blok kodu, który **zawsze** się wykona (np. do zwolnienia zasobów).
- `raise` – ręczne rzucanie (zgłaszanie) wyjątku.

Przykład `try...except...finally`

```
try:
    x = 1 / 0
except ZeroDivisionError as e:
    print(f"Błąd: {e}")
finally:
    print("Ten blok wykona się zawsze, niezależnie od wyniku powyżej.")
```

Tworzenie własnych wyjątków

```
class MojWyjatek(Exception):
    pass

def funkcja():
    raise MojWyjatek("Coś poszło nie tak!")

try:
    funkcja()
```

```
except MojWyjatek as e:
    print(e)
```

> **WAŻNE**:
> Własne wyjątki pozwalają lepiej organizować kod i odróżniać różne sytuacje wyjątkowe.

---

10. Iteratory i Generatory

Iteratory

- Obiekt jest **iteratorem**, jeśli posiada metody `__iter__()` i `__next__()`.
- `__next__()` zwraca kolejną wartość w sekwencji; gdy wartości się kończą, zgłaszany jest wyjątek `StopIteration`.

Generatory

- **Generator** to specjalna funkcja zawierająca słowo kluczowe `yield`.
- Za każdym razem, gdy napotka `yield`, zwraca wartość i „zamraża" stan funkcji. Przy kolejnym wywołaniu wraca do tego stanu i kontynuuje.

Przykład generatora

```
def generator_liczb(n):
    for i in range(n):
        yield i

gen = generator_liczb(5)
for liczba in gen:
    print(liczba)
# Output: 0 1 2 3 4
```

> **KLUCZOWE**:
> Generatory są bardzo wydajne przy pracy z dużymi zbiorami danych, bo nie muszą wszystkiego trzymać w pamięci na raz.

---

11. Abstrakcyjne Klasy Bazowe (ABC)

- **Klasy abstrakcyjne** w Pythonie umożliwiają zdefiniowanie interfejsu (metod), które muszą zostać zaimplementowane w klasach dziedziczących.
- Nie można tworzyć instancji klasy abstrakcyjnej (z założenia są niekompletne).

Przykład z `abc`

```
from abc import ABC, abstractmethod
```

```
class AbstrakcyjnaKlasa(ABC):
    @abstractmethod
    def metoda_abstrakcyjna(self):
        pass

class KonkretnaKlasa(AbstrakcyjnaKlasa):
    def metoda_abstrakcyjna(self):
        print("Implementacja metody abstrakcyjnej")

ob = KonkretnaKlasa()
ob.metoda_abstrakcyjna()
```

> **WAŻNE**:
> Jeśli klasa dziedziczy po klasie abstrakcyjnej i **nie** zdefiniuje wszystkich metod abstrakcyjnych, też staje się klasą abstrakcyjną.

---

12. Protokoły i interfejsy

W Pythonie nie ma formalnego słowa kluczowego `interface` jak w Javie, ale idea jest podobna:

- **Interfejs** to zbiór metod, które klasa powinna implementować, by „pasowała" do danego protokołu.
- Python opiera się na **kaczym typowaniu** – zamiast sprawdzać, czy obiekt „dziedziczy" z danej klasy, sprawdzamy, czy obiekt ma potrzebne metody.

> **Przykład**:
> Jeżeli obiekt posiada metody `__getitem__` i `__len__`, można go traktować jak **sekwencję** (np. listę, krotkę). Nie ma znaczenia, czy obiekt dziedziczy z `list` czy `tuple` – ważne, że ma odpowiednie metody.

---

13. Przeciążanie operatorów

W Pythonie możemy nadawać operatorom (+, -, *, /, itd.) nowe znaczenia dla naszych własnych klas. Robimy to poprzez zdefiniowanie odpowiednich „metod specjalnych":

- `__add__(self, other)` – obsługa operatora `+`
- `__sub__(self, other)` – obsługa operatora `-`
- `__mul__(self, other)` – obsługa operatora `*`
- ... i wiele innych (`__lt__`, `__gt__`, `__eq__`, `__str__`, `__repr__` itd.)

Przykład: Klasa, która „dodaje" się w inny sposób

```
class Liczba:
    def __init__(self, wartosc):
        self.wartosc = wartosc

    def __add__(self, other):
        if isinstance(other, Liczba):
            return Liczba(self.wartosc + other.wartosc)
```

```
        else:
            return NotImplemented

    def __repr__(self):
        return f"Liczba({self.wartosc})"

l1 = Liczba(5)
l2 = Liczba(7)
l3 = l1 + l2
print(l3)  # Liczba(12)
```

> **KLUCZOWE**:
> `__repr__` i `__str__` kontrolują sposób, w jaki obiekt jest wyświetlany (w konsoli i jako „reprezentacja" tekstowa).

---

14. Typy sekwencyjne i protokoły

Sekwencje w Pythonie (listy, krotki, łańcuchy znaków) charakteryzują się:

- Możliwością iterowania (for `elem in sekwencja`).
- Możliwością indeksowania (`sekwencja[i]`).
- Implementacją metod `__getitem__` i `__len__`.

Możesz też stworzyć własny typ sekwencji, implementując te metody.

Przykład własnej sekwencji

```
class MojaSekwencja:
    def __init__(self, dane):
        self.dane = dane

    def __getitem__(self, index):
        return self.dane[index]

    def __len__(self):
        return len(self.dane)

sek = MojaSekwencja([10, 20, 30])
print(sek[1])      # 20
print(len(sek))    # 3
for x in sek:
    print(x)
# 10, 20, 30
```

> **WAŻNE**:
> Dzięki temu obiekt `MojaSekwencja` zachowuje się jak typowa sekwencja (np. lista), bo implementuje protokół sekwencji Pythona.

---

15. Podsumowanie

1. **Obiekty** to fundament Pythona. Każdy obiekt ma tożsamość, typ, stan i zachowanie.
2. **Zmienne** są etykietami przypisanymi do obiektów (tylko obiekty mają typ).
3. **Typy danych** w Pythonie są dynamicznie sprawdzane w czasie wykonywania.
4. **Klasy** pozwalają tworzyć obiekty zdefiniowane przez atrybuty i metody.
5. **Atrybuty i metody** dzielą się na instancyjne i klasowe; mamy również dekoratory (`@staticmethod`, `@property`) rozszerzające funkcjonalność.
6. **Hermetyzacja** w Pythonie opiera się na konwencjach nazewniczych, ponieważ nie ma „prawdziwych" modyfikatorów dostępu.
7. **Dziedziczenie** umożliwia tworzenie nowych klas na podstawie już istniejących.
8. **Polimorfizm** daje możliwość pisania kodu, który działa dla różnych typów obiektów, jeśli mają one wymagane metody.
9. **Wyjątki** zapewniają bezpieczną i czytelną obsługę błędów.
10. **Iteratory i generatory** pozwalają wydajnie przetwarzać sekwencje danych.
11. **Klasy abstrakcyjne (ABC)** definiują interfejs, który musi być zaimplementowany w klasach dziedziczących.
12. **Protokoły i interfejsy** w Pythonie to konwencja: jeśli obiekt ma metody, które są wymagane, to „pasuje" do interfejsu.
13. **Przeciążanie operatorów** daje możliwość tworzenia bardziej zrozumiałych i przyjaznych interfejsów do własnych klas.
14. **Typy sekwencyjne** (listy, krotki, str, itp.) implementują `__getitem__()` i `__len__()`, co pozwala na łatwe iterowanie i indeksowanie.