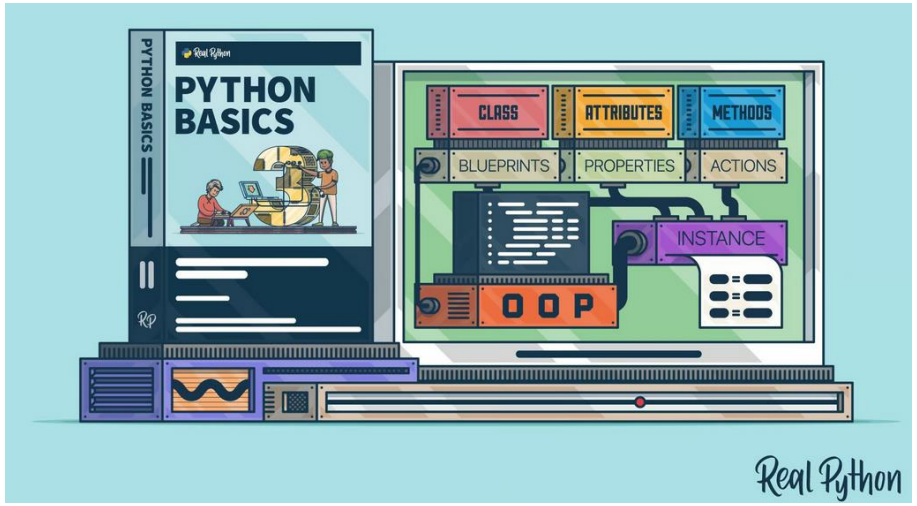
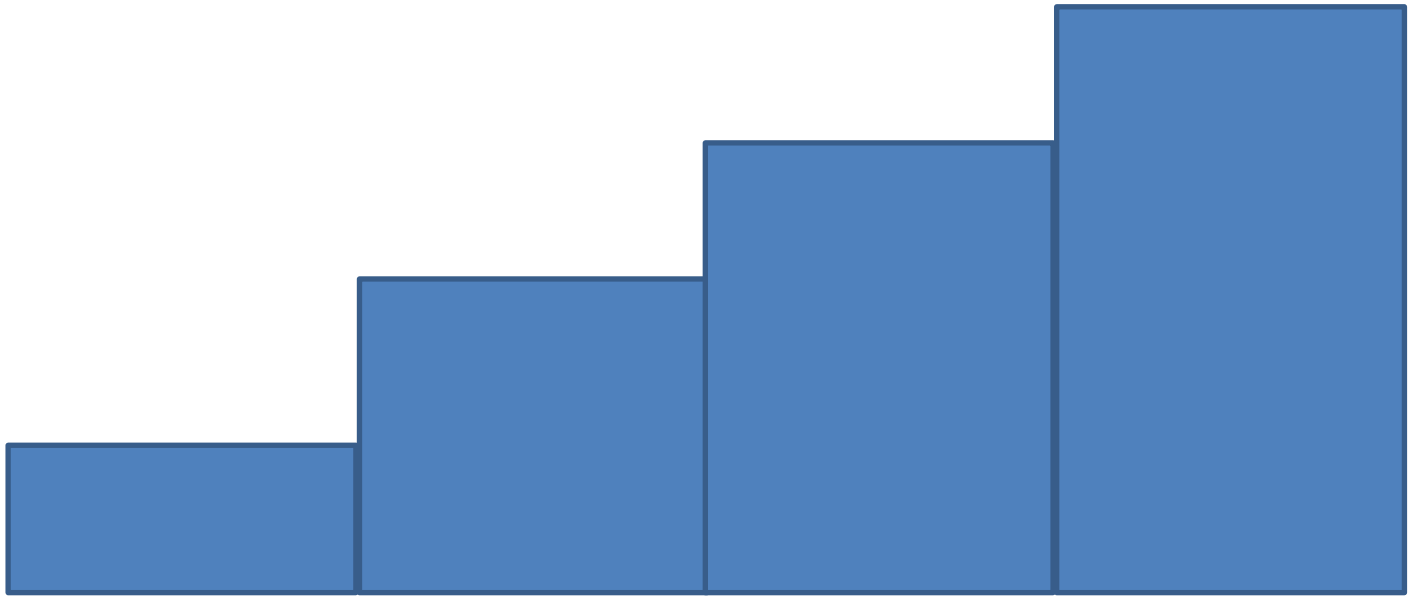


Programowanie obiektowe - wprowadzenie

M@я3k Pų𝔡€£kØ

Programowanie w Pythonie

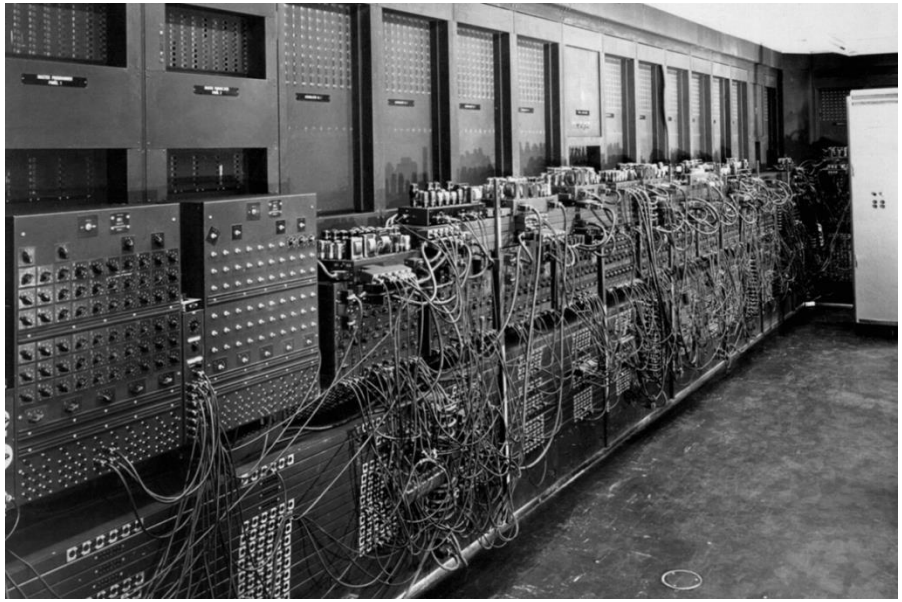
Spis treści



KOLEJNE GENERACJE PROGRAMOWANIA

Programowanie zero-jedynkowe

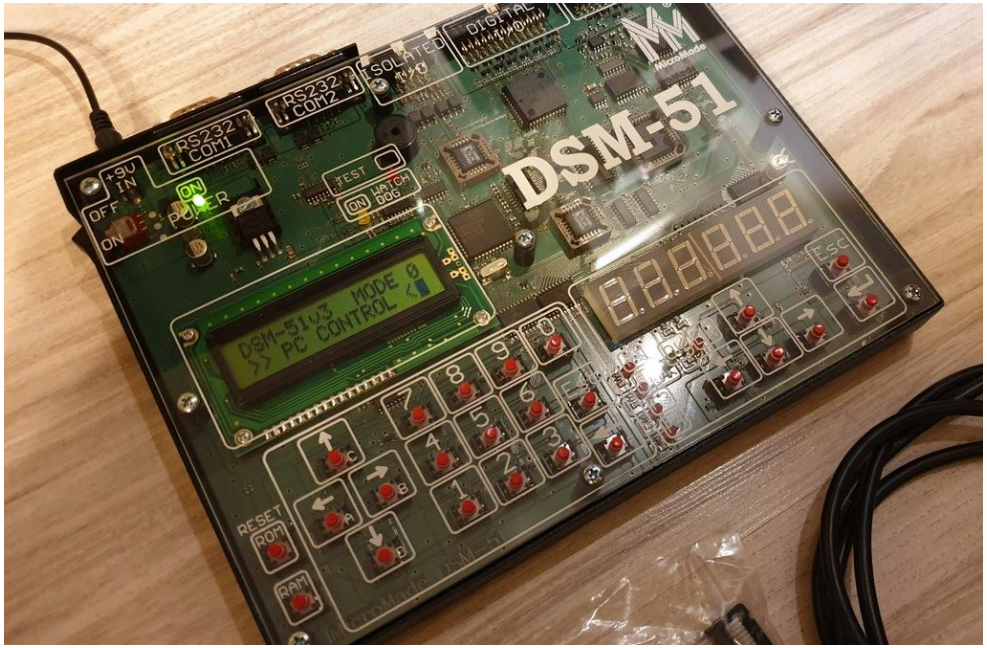
- Programowanie obiektowe
- 01010000 01110010 01101111 01100111
  01110010 01100001 01101101 01101111
  01110111 01100001 01101110 01101001
  01100101 00100000 01101111 01100010
  01101001 01100101 01101011 01110100
  01101111 01110111 01100101



https://i.gremicdn.pl/image/free/b356fe3f9e96ed2cc28c0997a27c2713

Programowanie w asemblerze

```
Z7kwadrati:
        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+8]
        imul    eax, [ebp+8]
        pop     ebp
        ret
```




```
main:
        push  ebp
        mov       ebp, esp
        and       esp, -16
        sub       esp, 32

        mov       [esp+28], 5
        mov       eax, [esp+28]
        push  eax
        call      _Z7kwadrati
        mov       [esp+24], eax

        mov       eax, [esp+28]
        imul      eax, [esp+28]
        mov       [esp+24], eax

        mov     eax, 0
        leave
        ret
```

Programowanie strukturalne

```
var i, j:Integer;
begin
 for i:=1 to 10 do
 begin
   Write(i : 3);
   for j:=1 to 10 do
    Write(j : 3);
   Writeln;
 end;
end.
```

https://livingcomputers.org/getattachment/70132c0b-839f-47cf-a69e-6d0c894f76fa/attachment.aspx

PROGRAMOWANIE OBIEKTOWE

Klasa a obiekt

- **Klasa** – to szablon, projekt
- **Obiekt** – to element stworzony według szablonu jakim jest klasa

Klasa
(projekt domu)

Obiekty tej klasy
(budynek zbudowany według tego projektu)

Klasy

- Klasa to definicja formatu danych według którego tworzone są obiekty.

- Klasa opisuje jakie dane może zawierać obiekt oraz jakim operacjom może zostać poddany lub jakie operacje może sam wykonać.

Obiekty

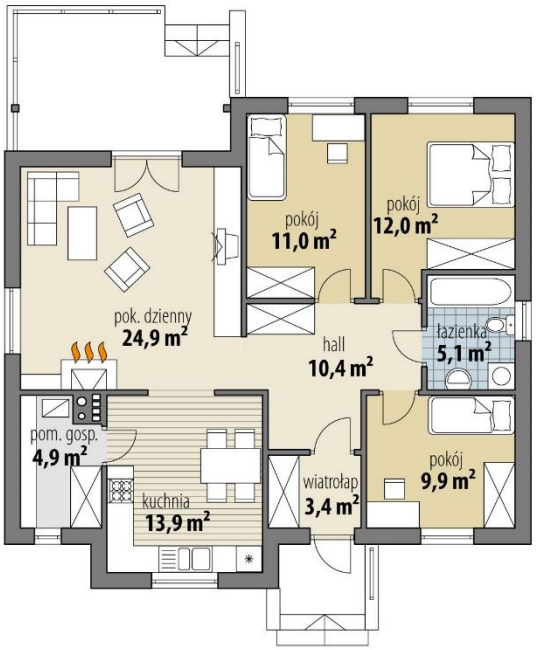
- Obiekt jest po prostu zmienną.

- Obiekt ≠ prosty typ (`int`, `float`, `char`).
- Obiekt > struktura (`struct`)

- Obiekt – rozbudowany typ danych.
  - Definiowany przez użytkownika
  - Własny typ obiektów można utworzyć za pomocą klas
  - Zawiera dane i operacje jakie może wykonać

Klasa i obiekt

Klasa	Obiekt
Szablon (projekt)	Element utworzony według szablonu
Twór abstrakcyjny	Konkretny, istniejący byt
Przechowuje cechy	Zawiera konkretne wartości
Deklarowana tylko raz	Może ich być utworzona dowolna ilość



Klasa i obiekt - definiowanie

- Pojęcie klasa i obiekt pozwala na utworzenie zmiennych szczególnego typu
- **Klasa** - typ zmiennej
- **Obiekt** – zmienna (instancja klasy)

Definicja klasy	`class` **`nazwa_klasy`**
Utworzenie obiektu danej klasy	***`tworzony_obiekt`*** **`= nazwa_klasy()`**

Deklarowanie klasy

```
class Dom:
    Powierzchnia = 100
    Cena = 1000000
    IloscPokoi = 6
```

Definicja klasy
Słowo kluczowe `class` i nazwa klasy `Dom`:

Parametry klasy `Dom` wraz z wartościami inicjalizującymi.
Są to jednocześnie wartości domyślne, które będą zapisane w powstałym obiekcie jeśli nie zostaną wprowadzone inne.

Deklarowanie klasy

```
class Dom:
    Powierzchnia = 100
    Cena = 1000000
    IloscPokoi = 6
    IloscPieter = 2
    Garaz = True
    Piwnica = False
    Ogrod = True
```

Definicja klasy
Słowo kluczowe `class` i nazwa klasy `Dom`:

Parametry klasy `Dom` wraz z wartościami inicjalizującymi.
Są to jednocześnie wartości domyślne, które będą zapisane w powstałym obiekcie jeśli nie zostaną wprowadzone inne.

Klasa pusta

- Chcąc utworzyć pustą klasę, należy w jej treść dodać słowo kluczowe `pass`.

```
class Dom:
   pass
```

ATRYBUTY I METODY

Atrybuty

- Atrybuty (pola) to cechy, które będzie posiadał obiekt danej klasy.
- W praktyce to zmienne mające jakąś wartość i przechowywane w danej klasie lub obiekcie.

- Przykład

```
class Dom:
   Powierzchnia = 100
   Cena = 1000000
```

Atrybuty w Pythonie

- **atrybuty klasy** – *atrybuty statyczne*
- Przechowują dane specyficzne dla klasy.
- Wspólne dla wszystkich obiektów.

- **atrybuty instancji** (obiektu) – *atrybuty dynamiczne*
- Określają, jakie cechy posiada dany obiekt.
- Mogą być inne dla każdego obiektu tej samej klasy, co pozwala je odróżnić.

Atrybuty klasy i instancji

- **atrybuty klasy** – (atrybuty statyczne)

```
class Dom:
   Powierzchnia = 100
   Cena = 1000000
```

- **atrybuty instancji** – (atrybuty dynamiczne)

```
class Dom:
     def __init__(self, powierzchnia, cena):
          self.powierzchnia = powierzchnia
          self. cena = cena
```

Atrybuty klasy

- **atrybuty klasy** – (atrybuty statyczne)
- Przechowują dane specyficzne dla klasy. Tworzy się je najczęściej bezpośrednio po nazwie klasy bez użycia przedrostka *self*.
- Wszystkie obiekty mają je takie same.
- Można korzystać z nich zarówno z poziomu obiektu jak i bez posiadania jakiegokolwiek obiektu danej klasy poprzez notację z kropką: *Klasa.atrybut*.

```
class Dom:
    powierzchnia = 100
    cena = 10000000
    pokoje = 4

print (Dom.powierzchnia)   #100
print (Dom.cena)           #10000000
print (Dom.pokoje)         #4
```

Atrybuty instancji

- **atrybuty instancji** – (Atrybuty dynamiczne)
- Odnoszą się do instancji (obiektów) i określają, jakie cechy dany obiekt posiada. Atrybuty mogą być inne dla każdego obiektu tej samej klasy. Pozwala to odróżnić od siebie wiele obiektów tej samej klasy.
- Tworzy się je zazwyczaj w konstruktorze *__init__* (inaczej: metodzie inicjalizacji), a ich nazwy są poprzedzone przedrostkiem *self*.
  - *“self” to parametr odnoszący się do instancji klasy. Informuje, że dany atrybut lub metoda dotyczy właśnie instancji i może zmieniać jej stan.*
  - *“self” nie jest słowem kluczowym, a jedynie powszechnie przyjętą konwencją, dlatego można je zastąpić dowolnym słowem. Nie jest to zalecane, ponieważ może wprowadzać w błąd.*
- Odwołanie do atrybutów instancji poza ciałem klasy odbywa się jedynie z wykorzystaniem obiektu. Nie ma możliwości np. zmiany wartości atrybutu instancji bez posiadania instancji tej klasy:
  - nie istnieje obiekt → nie istnieją jego atrybuty.

Atrybuty instancji

```
   class Dom:
      def __init__(self, powierzchnia, cena):
        # instance attributes
        self.powierzchnia = powierzchnia
        self. cena = cena

Willa = Dom (120, 20000000)

print (Dom.pokoje)              #4
print (Dom.powierzchnia)        #10000000


print (Willa.pokoje)            #4
print (Willa.powierzchnia)      #120
print (Willa.cena)              #20000000
```

Metody

- Metody to funkcje zdefiniowane wewnątrz klasy.
- Definiują zachowanie, umożliwiając im wykonywanie konkretnych zadań,
- Nie można ich definiować poza ciałem klasy.
- Deklaracja:
  - modyfikator
  - typ zwracany przez metodę
  - nazwa metody (NazwaMetody())
  - lista typów i nazw parametrów metody

Metody

- **metody instancji** – dotyczą wszystkich tworzonych obiektów i wpływają na ich stan (np. modyfikują ich atrybuty).
- **metody statyczne** – choć są częścią klasy, to nie widzą innych jej elementów i nie muszą mieć z klasą żadnego związku (są niezależne).
- **metody klasy** – pracują na poziomie klasy. Nie wymagają instancji, ale mogą odwoływać się do pozostałych jej metod (również statycznych).
- **metody specjalne** (ang. *special methods, dunder methods* od *“double underscore”*) – metody rozpoczynające się oraz zakończone podwójnym znakiem podkreślenia oraz wywoływane przez Pythona w określonych sytuacjach.

Metody instancji

- **metody instancji** – (podobnie jak w przypadku atrybutów instancji) dotyczą wszystkich tworzonych obiektów i wpływają na ich stan (np. modyfikują ich atrybuty).
- Tworząc metodę instancji, jako pierwszy parametr zawsze przekazuje się *self*, a dopiero po nim kolejne. *self* wskazuje jedynie, że odnosi się do instancji i nie przekazuje się jego wartości podczas wywoływania danej metody.

```
class Dom:
      pokoje = 4
      def __init__(self, powierzchnia, cena):
            self.powierzchnia = powierzchnia
            self.cena = cena
      def WartoscPokoju(self):
              avercost = self.cena/self.pokoje
              return (avercost)

Willa = Dom (100, 1000000)
print (Dom. WartoscPokoju(Willa), „ za pokoj w willi")
```

Metody statyczne

- **metody statyczne** – (pomimo bycia częścią klasy) nie widzą innych elementów klasy i nie muszą mieć z klasą żadnego związku (są niezależne).
- Nie odwołują się do obiektów, dlatego nie posiadają parametru *self*.
- Ich definicja poprzedzana jest dekoratorem *@staticmethod*, a wywołanie odbywa się poprzez nazwę klasy, w której się zawierają (*Klasa.metoda_statyczna()*).

```
class Dom:
      pokoje = 4
      def __init__(self, powierzchnia, cena):
            self.powierzchnia = powierzchnia
            self.cena = cena
      @staticmethod
      def IloscZlotych(kurs, dolary):
            zlotowka = kurs * dolary
            return zlotowka
Willa = Dom (100, 1000000)
print (Dom.IloscZlotych (4.5, 100), " zlotych")
```

Metody klasy

- **metody klasy** – pracują na poziomie klasy. Podobnie jak metody statyczne, nie wymagają instancji, ale są świadome bycia częścią klasy i mogą odwoływać się do pozostałych jej metod (również statycznych).
- Tworzy się je z użyciem dekoratora *@classmethod* oraz poprzez przekazanie parametru *cls*.
  - *Parametr "cls", podobnie jak "self", jest konwencją nazewniczą. Odnosi się do klasy i jest używany wraz z dekoratorem @classmethod. Umożliwia dostęp do atrybutów i metod klasy, dzięki czemu modyfikuje jej stan (nie wpływa jednak bezpośrednio na stan instancji!).*

```
class Dom:
      pokoje = 4
      def __init__(self, powierzchnia, cena):
            self.powierzchnia = powierzchnia
            self.cena = cena
       @classmethod
       def IloscLudzi (cls, osoby):
            zaludnienie = osoby/cls.pokoje
            return zaludnienie
Willa = Dom (100, 1000000)
print (Willa.IloscLudzi (10), "osob w pokoju")
```

Metody specjalne

- **metody specjalne** (ang. *special methods, dunder methods* od *“double underscore”*) – są zwane również metodami magicznymi.
- Rozpoczynają się i kończą podwójnym znakiem podkreślenia. Są wywoływane przez Pythona w określonych sytuacjach.
- Metody specjalne mogą odwoływać się do obiektu dzięki parametrowi *self* lub do samej klasy za pomocą *cls*.

```
class Dom:
      pokoje = 4
      def __init__(self, powierzchnia, cena):
            self.powierzchnia = powierzchnia
            self.cena = cena
Willa = Dom (100, 1000000)
```

Metody specjalne - przykłady

Tworzenie i usuwanie instancji	
__new__(cls, ...)	Metoda wywoływana jako pierwsza podczas tworzenia nowych obiektów. Jej zadaniem jest stworzenie i zwrócenie nowego obiektu danej klasy, który zaraz potem jest przekazywany do metody **__init__** jako pierwszy argument (self).
__init__(self, ...)	Jej zadaniem jest inicjalizacja obiektu z pomocą argumentów przekazanych przy wywołaniu klasy. Metoda często nazywana jest konstruktorem, którym tak właściwie jest razem z metodą **__new__.**
__del__(self)	Metoda usuwająca obiekt z pamięci (kasująca go). Wywoływana jest w momencie gdy obiekt kończy swój cykl życia i za zadanie ma zrobienie porządku po nim.
Reprezentacja obiektów	
__str__(self)	Metoda wywoływana przy konwersji obiektu do stringa str(x) , oraz przy wywoływaniu funkcji print(x).
__repr__(self)	Metoda zwróca oficjalną reprezentację, jej wynik można zobaczyć wywołując funkcję repr()
__hash__(self)	Metoda __hash__ wykorzystywana jest do haszowania obiektu, czyli zamiany jego wartości na wartość liczbową. Jej implementacja jest potrzebna dla ustalenia zachowania obiektów w roli kluczy słowników.

Przeciążanie metod

- Przeciążanie metod polega na utworzeniu kilku metod o tej samej nazwie, ale o różnej ilości argumentów
  - Przeciążanie ≠ przesłanianie
  - przesłanianie - nadpisywanie metody w klasie potomnej.
- W pythonie nie ma typowego przeciążania metod (jak w C++ czy Javie)
  - Tworząc nową metodę o tej samej nazwie w danej klasie powoduje się, że wcześniejsza metoda przestaje być używana.

Przeciążanie metod – rozwiązania

1. Podanie listy argumentów jako funkcji

Lista jest dynamiczną strukturą. Tę samą funkcję można wywoływać z różna ilością parametrów i różnych typów.

*args - przekazanie do funkcji dowolną liczbę argumentów pozycyjnych (*tuple*)

```
def add(*args):
   return sum(args)
>>> add(1, 2)
3
>>> add(1, 2, 3)
6
>>> add(1, 2, 3, 4)
10
```

2. Zastosowanie argumentów domyślnych i opcjonalnych

Jeśli opcjonalnych brak, można zadeklarować inne zachowanie metody

Ta metoda najlepiej przydaje się przy użyciu 1 lub 2 argumentów. Większa ilość jest możliwa przy zachowaniu odpowiedniej dyscypliny programisty.

```
def add(arg1, arg2=None):
  if arg2 is not None:
     return arg1 + arg2
  else:
     return arg1 ** 2
```

Tworzenie nowych obiektów

- Nowy obiekt danej klasy tworzymy przypisując jej nazwę klasy. Korzysta się tu z wzoru:

***tworzony_obiekt* = nazwa_klasy()**

- Przykłady

```
Domek = Dom ()
Willa = Dom ()
Dacza = Dom ()
```

- Nie można na raz utworzyć kilku obiektów tej samej klasy

~~`Willa1, Willa2 = Dom ()`~~

Przykład – program ''Klasa-punkt''

```
'''
Klasa-punkt
'''
import math

class Punkt:
    x = 0
    y = 0

    def __init__(self, x, y):      # instance attributes
        self.x = x
        self.y = y

    def przesun (self, a, b):
        self.x = self.x + a
        self.y = self.y + b

    def odleglosc_0_0 (self):
        d = math.sqrt (self.x*self.x+self.y*self.y)
        return d
```

```
print (Punkt.x)
print (Punkt.y)

A = Punkt (5, 12)
print (A.x)
print (A.y)
print (A.odleglosc_0_0 ())

A.przesun (11, 11)
print (A.x)
print (A.y)
print (A.odleglosc_0_0 ())

Punkt.x = 10
Punkt.y = 10

B= Punkt (Punkt.x, Punkt.y)
print (B.x)
print (B.y)
print (B.odleglosc (3,4))
```