**I.1) Co to są moduły i pakiety? Jak je wykorzystujemy w projekcie?**Moduł – to po prostu plik z rozszerzeniem .py zawierający jakieś funkcje  
Pakiet – jest to zbiór modułów (zebranych w folder)  
Moduły i pakiety przyśpieszają tworzenie kodu przez wykorzystywanie już gotowych rozwiązań. Odwołujemy się do nich przez:  
 import modul #zaimportowanie danego modułu  
 import pakiet.modul #zaimportowanie danego modułu z pakietu  
 from pakiet import modul #zaimportowanie i dodanie do przestrzeni nazw  
  
**I.2) Obiekty i klasy:**  
**a) Jak definiujemy klasy?**  
 class MojaKlasa:

zmienna = "blah"  
**b) Jak tworzymy obiekty danej klasy?** mojobiekt = MojaKlasa()  
 mojobiekt.zmienna  
**c) Co to są metody? Jak je definiujemy i jak je stosujemy na obiektach?**Metody to funkcje które są w klasie:  
 class MojaKlasa:

def metoda(self, a, b): #self jest potrzebne

print(a)  
 print(b)

obiekt = MojaKlasa()  
 obiekt.metoda(1,”dwa”) #printuje a i b  
**d) Jak odwołujemy się do atrybutów (pól i metod) obiektu?**Dodajemy przed nazwę pola/zmiennej nazwę obiektu:  
 (nazwa obiektu).(nazwa zmiennej / metody)  
**e) Co to są metody specjalne (np. \_\_init\_\_, \_\_add\_\_, \_\_str\_\_ itp.)?**Metody specjalne służą do przeciążania operatorów.\_\_init\_\_() - to metoda która zostanie wykonana automatycznie zaraz po utworzeniu nowej instancji  
\_\_add\_\_() – obsługuje wykonanie operatora + na obiekcie

\_\_str\_\_() – definiuje co ma zwracać obiekt przy printowaniu go

\_\_repr\_\_() – definiuje co ma zwracać obiekt przy wywołaniu go w shellu  
#https://pl.python.org/docs/ref/node15.html

Nie powinniśmy sami tworzyć nowych metod zaczynających się od \_\_  
**f) Co to jest dziedziczenie i polimorfizm? Jak to działa?**Dziedziczenie – czasami musimy zdefiniować różne klasy, które mają wspólne cechy – np. wszystkie zwierzęta mogą mieć jakiś wiek, wymiary itd.

Polimorfizm – Zachowanie funkcji zależy od tego, co do niej przekażemy lub w jaki sposób ją wywołamy (pole kwadratu, pole okręgu, pole trójkąta) opisywane przez wspólną funkcję pole()

Dziedziczenie i polimorfizm:  
 class Figura:

pi = 3.14

wspSkali = 5  
 def pole(self): #zwraca pole koła  
 return self.pi\*self.promien\*\*2 \*self.wspSkali

class Okrag(Figura): #funkcja na pole jest dziedziczona

promien = 1

class Kwadrat(Figura):

bok = 2

def pole(self): #redefiniuje metodę pole

return self.bok\*\*2 \*self.wspSkali

#wsp skali jest dziedziczony z figury, pole jest „nadpisywane”

okr = Okrag()

kwadr = Kwadrat()

print(okr.pole())

print(kwadr.pole())

**II.1 a)**  
 Podaj liczbę:5

Pierwiastek liczby 5.0 = 2.23606797749979

jej logarytm = 1.6094379124341003

Obwód kola o promieniu 5.0 = 31.41592653589793  
  
**c)**Odległość punktu od układu wsp - math.hypot(\*coordinates)  
  
**d) Szybkość działania różnych funkcji**

**1. Błąd – w 7 linijce, (drugi return)**  
Podaj liczbę całkowitą:5

5!, wynosi rekurencyjnie: 120

5!, wynosi iteracyjnie: 120

5!, wynosi math.factorial: 120

Obliczenia rekurencyjnie trwały: 0.0 sekund

Obliczenia iteracyjnie trwały: 0.0 sekund

Obliczenia math.factorial trwały: 0.0 sekund  
  
100!, wynosi rekurencyjnie: … #(wszystkie równe) 158 znaków

100!, wynosi iteracyjnie: …

100!, wynosi math.factorial: …

Obliczenia rekurencyjnie trwały: 0.0 sekund

Obliczenia iteracyjnie trwały: 0.0 sekund

Obliczenia math.factorial trwały: 0.0 sekund

Podaj liczbę całkowitą:1024 (maksymalna wartość dla rekurencji)

1024!, wynosi rekurencyjnie: … #(nadal wszystkie równe) 2640 znaków…

1024!, wynosi iteracyjnie: …

1024!, wynosi math.factorial: …

Obliczenia rekurencyjnie trwały: 0.0010001659393310547 sekund

Obliczenia iteracyjnie trwały: 0.0009992122650146484 sekund

Obliczenia math.factorial trwały: 0.0 sekund

Podaj liczbę całkowitą:50000

50000!, wynosi iteracyjnie: … (1839 **linii** kodu)

50000!, wynosi math.factorial: …

Obliczenia iteracyjnie trwały: 1.6870005130767822 sekund

Obliczenia math.factorial trwały: 0.19900083541870117 sekund  
  
Podaj liczbę całkowitą:100000

100000!, wynosi iteracyjnie: … (3936 linii kodu)

100000!, wynosi math.factorial: …

Obliczenia iteracyjnie trwały: 8.884720087051392 sekund

Obliczenia math.factorial trwały: 0.6660048961639404 sekund  
  
Podaj liczbę całkowitą:200000 #wypisywanie wyników trwa dłużej

Obliczenia iteracyjnie trwały: 39.30905604362488 sekund

Obliczenia math.factorial trwały: 2.2111241817474365 sekund  
  
Wniosek: najszybsze jest math.factorial, a najwolniejsza rekurencja.

**II.2 a)**  
\_\_init\_\_() – przypisze domyślnie x i y do 1  
dlugosc() – zwróci odległość punktu od środka układu współrzędnych  
\_\_add\_\_() – zdefiniuje zachowanie gdy dodajemy dwa wektory  
\_\_str\_\_() i \_\_repr\_\_() – definiuje jak wektor ma być wyświetlany gdy jest printowany.  
  
**b)**

Zakładając że fragment zawierający klasę wektor będzie fragmentem skryptu wynikiem tego skryptu będzie błąd: nigdy nie importujemy modułu math.  
Ale po imporcie:  
wektor w1= Wektor(1,1) ma długość 1.4142135623730951

wektor w2= Wektor(2,2) ma długość 2.8284271247461903

wektor w3= Wektor(3,3) ma długość 4.242640687119286