Laboratorium **Programowanie w języku Python 2** Wydział Elektrotechniki Automatyki I Informatyki Politechnika Świętokrzyska	
Studia: **Stacionarne I stopnia**	Kierunek: **Informatyka**
Data wykonania: **01.04.2021**	Grupa: **3ID16B**
Imię I nazwisko: **Arkadiusz Więcław**	Temat ćwiczenia: **Dziedziczenie wielokrotne** **Dynamiczne tworzenie klas** **Klasy abstrakcyjne**

Zad 1:

```
from abc import ABC, abstractmethod
def p1():
    """
    Przyklad tworzy klas Person ktorej konstruktor przyjmuje dwa
argumenty .Dodatkowo klasa zawiera metode
    printname ktora wypisuje na ekran przyjete w konstruktorze
argumenty
    """

    class Person:
        def __init__(self, fname, lname):
            self.firstname = fname
            self.lastname = lname


        def printname(self):
            print(self.firstname, self.lastname)

    p1 = Person("John", "Doey")
    p1.printname()
    p2 = Person("Gary", "God")
    p2.printname()


def p2():
    """
    Przyklad jest podobny do poprzedniego  Student ktora dziedziczy po
klasie
    Person.
    """

    class Person:
        def __init__(self, fname, lname):
            self.firstname = fname
            self.lastname = lname

        def printname(self):
            print(self.firstname, self.lastname)
```

```
    class Student(Person):
        pass

    x = Student("Mike", "Olsen")
    x.printname()



def p3():
    """
    Przyklad tworzony pusta klasa  test ktora dziedzicy po klasie
object , potem tworzy na jej
    podstawie obiekt t ktoremu przypisywana jest wlasciwosc  o nazwie
a przyjmuja wartosc
    """

    class test(object):
        pass

    t = test()
    t.a = 10
    print(t.a)
    print(dir(test))
    t.b = 20
    t.c = "www"
    print(t.b)
    print(t.c)
    print(dir(test))


def p4():
    """
    Przyklad tworzy klasy pusta Foo  w sposob dynamiczny wykorzystujac
funkcje type()
    o na koncu jest  wyswieltane obiekt klasy
    """
    Foo = type('Foo', (), {})
    x = Foo()
    Kawa = type('Kawa', (), {})
    Naz = type('Naz', (), {})
    print(x)
    y = Kawa()
    z = Naz()
```

```
    print(y)
    print(x)


def p5():
    """
    Przyklad tworzy dwie klasy (Foo, Bar) za pomoca type() ,jedna
klasa dziedzicy po drugie
    Klasa Bar ma zdefiniowany wlasciwosc attr przyjmujaca wartosc
100.Na koncu jest tworzony obiekt x
    a nastepnie jest wyswietlany zawartosc attr a potem jaka klasa
jest potomna a nastpnie jaka klasa
    jest bazowa Dodalem dodatkowy atrybut do klas Bar, utworzylem dwie
klasy za pomoca
    type jedna dziedziczy po drugiej i po klasie Bar
    """
    Foo = type('Foo', (), {})
    Bar = type('Bar', (Foo,), dict(attr=100, ww=200))
    x = Bar()
    print(x.attr)
    print(x.ww)
    print(x.__class__)
    print(x.__class__.__bases__)
    Klas = type('Klas', (), dict(woka="sss", poka="qaw"))
    Kub = type('Kub', (Klas, Bar,), {})
    test = Klas()
    print(test.woka)
    print(test.__class__)
    print(test.__class__.__bases__)
    print(Klas.__subclasses__())


def p6():
    """
    Przyklad tworzy dwie klasy jedna dziedziczy po drugiej.Klasa
bazowa w konstruktorze ma parametr mamalName
    i wewnatrz konstruktora  jest wyswietlany text wraz z wprowadzonym
wartoscia w argumencie w
    tym przypadku wartosc jest pusta Nastepnie klasa potomna dziedzica
po klasie bazowej  i wykorzystujac
    funkcje super() odwolujemy sie do klasy bazowej

    """
```

```
    class Mammal(object):
        def __init__(self, mammalName):
            print(mammalName, 'is a warm-blooded animal.')

    class Dog(Mammal):
        def __init__(self):
            print('Dog has four legs.')
            super().__init__('Dog')

    class Jamnik(Dog):
        def __init__(self):
            super().__init__()
            print(" klasa Jamnik")

    w = Jamnik()


def p7():
    """
    Przyklad tworzy 3 klasy klasa dziedzidzy po dwoch klasach A i B .
Za pomoca fukncji super() moze zobaczyc
    w jakiej kolejnosc beda wywolywane metody  klas
    Dodalem dodatkowa klase D po ktorej bedzie dziedzic C . Umiescilem
to klasa jako druga  po ktorej
    bedzie dziedziczyla klasa C. Co spowodowalo ze metody klasy D jest
wywolana jako druga
    """

    class A(object):
        def __init__(self):
            super(A, self).__init__()
            print('init A')

    class B(object):
        def __init__(self):
            super(B, self).__init__()
            print('init B')

    class D(object):
        def __init__(self):
            super(D, self).__init__()
            print('init D')
```

```
    class C(A, D, B):
        def __init__(self):
            super(C, self).__init__()
            print('init C')

    c = C()


def p8():
    """
    Przyklad pokazuje jakiej kolejnosc bede wywolywane metoda klas
    Dodalem dodatkowa klasa po ktorej bedzie dziedziczyla klasa D
    """

    class A(object):
        def foo(self, call_from):
            print("foo from A, call from %s" % call_from)
            super().foo("A")

    class B(A):
        def foo(self, call_from):
            print("foo from B, call from %s" % call_from)
            super().foo("B")

    class C(A):
        def foo(self, call_from):
            print("foo from C, call from %s" % call_from)
            super().foo("C")

    class F(object):
        def foo(self, call_from):
            print("foo from F, call from %s" % call_from)

    class G(object):
        def foo(self, call_from):
            print("foo from G, call from %s" % call_from)

    class D(B, C, F, G):
        def foo(self):
            print("foo from D")
            super().foo("D")
```

```
    d = D()
    d.foo()
    print(D.__mro__)


def p9():
    """
    Przyklad tworzy pseudoklasa abstrakcyjna
    która ma metoda ktora nic nie robi . Po tej klasie dziedziczy
klasa B
    Dodalem kolejna klase ktora tez bedzie dziedzic po klasie
AbstractClassExample
    """

    class AbstractClassExample():
        def do_something(self):
            pass

    class B(AbstractClassExample):
        pass

    class D(AbstractClassExample):
        pass

    a = AbstractClassExample()
    b = B()
    d = D()
    print("Klasa nic nie robi ")


def p10():
    """
    Przyklad tworzy klasa abstrakcyjna i nastepnie dwie klasy
dziedziczace po niej  jedna dodaje do zmiennej
    value 42  o druga mnozy wartosc zmiennej valueo 42
    Dodalem  kolejna klasa ktore dziedziczy po klasie abstrakcyjnej
AbstractClassExample ktore odejmuje 42
    """

    class AbstractClassExample():
        def __init__(self, value):
            self.value = value
            super().__init__()
```

```
        @abstractmethod
        def do_something(self):
            pass

    class DoAdd42(AbstractClassExample):
        def do_something(self):
            return self.value + 42

    class DoMul42(AbstractClassExample):
        def do_something(self):
            return self.value * 42

    class DoSub20(AbstractClassExample):
        def do_something(self):
            return self.value - 42

    x = DoAdd42(10)
    y = DoMul42(10)
    z = DoSub20(20)
    print(x.do_something())
    print(y.do_something())
    print(z.do_something())


def p11():
    """
    Przyklad tworzy klase z abstrakcyjna z metoda abstrakcyjna
do_something
    klasa anaotherSubclass dziedzicy po klasie abstraykcyjne i tworzy
wlasna wersje metody do-someting
    w ktorej wywoluje wersje metody z klasy abstrakcyjne za pomoca
funkcji super() dodatkowa wypsuje
    komunikat
    Dodalem kolejna metode abstrakcyjna i jej implementacje
    """

    class AbstractClassExample():
        @abstractmethod
        def do_something(self):
            print("Implementacja metody abstrakcyjnej ")
```

```
    class AnotherSubclass(AbstractClassExample):
        def do_something(self):
            super().do_something()
            print("Metoda dziedzicząca")


    x = AnotherSubclass()
    x.do_something()


def main():
    print("\nPrzyklad 1 =")
    p1()
    print("\nPrzyklad 2 =")
    p2()
    print("\nPrzyklad 3 =")
    p3()
    print("\nPrzyklad 4 =")
    p4()
    print("\nPrzyklad 5 =")
    p5()
    print("\nPrzyklad 6 =")
    p6()
    print("\nPrzyklad 7 =")
    p7()
    print("\nPrzyklad 8 =")
    p8()
    print("\nPrzyklad 9 =")
    p9()
    print("\nPrzyklad 10 =")
    p10()
    print("\nPrzyklad 11 =")
    p11()


main()
```

Wyniki:

```
Przyklad 1 =

John Doey

Gary God


Przyklad 2 =

Mike Olsen


Przyklad 3 =

10

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']

20

www

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']


Przyklad 4 =

<__main__.Foo object at 0x0000026E6634FEB0>

<__main__.Kawa object at 0x0000026E6634FE80>

<__main__.Foo object at 0x0000026E6634FEB0>
```

```
Przyklad 5 =
100
200
<class '__main__.Bar'>
(<class '__main__.Foo'>,)
sss
<class '__main__.Klas'>
(<class 'object'>,)
[<class '__main__.Kub'>]

Przyklad 6 =
Dog has four legs.
Dog is a warm-blooded animal.
klasa Jamnik

Przyklad 7 =
init B
init D
init A
init C

Przyklad 8 =
foo from D
foo from B, call from D
foo from C, call from B
foo from A, call from C
foo from F, call from A
```

```
(<class '__main__.p8.<locals>.D'>, <class '__main__.p8.<locals>.B'>,
<class '__main__.p8.<locals>.C'>, <class '__main__.p8.<locals>.A'>,
<class '__main__.p8.<locals>.F'>, <class '__main__.p8.<locals>.G'>,
<class 'object'>)


Przyklad 9 =

Klasa nic nie robi


Przyklad 10 =

52

420

-22


Przyklad 11 =

Implementacja metody abstrakcyjnej

Metoda dziedzicząca
```

Zad 2:

```
class MojaClasa:
       pass
pusta = MojaClasa()
print("\nPrzyklad 1 =")
print(dir(pusta))
print(pusta.__class__)
print("Metoda __getattribute__ " ,pusta.__getattribute__)
"""
Wywoływane bezwarunkowo w celu zaimplementowania dostępu do atrybutów
dla wystąpień klasy.
Aby uniknąć nieskończonej rekurencji w tej metodzie, jej implementacja
powinna zawsze wywoływać metodę klasy bazowej o
tej samej nazwie, aby uzyskać dostęp do potrzebnych jej atrybutów, na
przykład obiekt
"""
print("Metoda__setattr__ " , pusta.__setattr__)
"""
Ta metoda jest wywoływana zamiast normalnego mechanizmu (tj.
```

```
Przechowuje wartość w słowniku instancji).
Jeśli __setattr __ () chce przypisać atrybut instancji, nie powinien
po prostu wykonywać
self.name = wartość - spowodowałoby to rekurencyjne wywołanie samego
siebie.
"""

"""
Ta metoda zwraca rozmiar obiektu
"""
print("Metoda __sizeof__() " ,pusta.__sizeof__())
"""
Ta metoda jest wywoływana, gdy funkcja print () lub str () jest
wywoływana na obiekcie.
Ta metoda zwracać obiekt String. Jeśli nie zaimplementujemy funkcji
__str __ () dla klasy, wtedy
używana jest wbudowana implementacja obiektu, która faktycznie
wywołuje funkcję __repr __ ().
"""
print("Metoda __str__() ",pusta.__str__())
"""
Zwraca reprezentację obiektu w formacie łańcucha.i
Ta metoda jest wywoływana, gdy funkcja repr () jest wywoływana na
obiekcie.
"""
print("Metoda __repr__()", pusta.__repr__())
"""
Zwraca wszystkie zdefiniowa atrybut w postaci slownika
(atrybut:wartosc)
"""
pusta.c = 5
print("Atrybut __dict__ ",pusta.__dict__)
"""
__delattr__() - usuwa atrybut
__setattr__() - ustawia atrybut
"""
```

Wynik:

```
Przyklad 1 =

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
```

```
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']

<class '__main__.MojaClasa'>

Metoda __getattribute__  <method-wrapper '__getattribute__' of
MojaClasa object at 0x0000026E661212E0>

Metoda__setattr__  <method-wrapper '__setattr__' of MojaClasa object
at 0x0000026E661212E0>

Metoda __sizeof__()  32

Metoda __str__()  <__main__.MojaClasa object at 0x0000026E661212E0>

Metoda __repr__() <__main__.MojaClasa object at 0x0000026E661212E0>

Atrybut __dict__  {'c': 5}
```

Zad 3:

```
"""
Bez problemu można się odwolywac do pierwszej klasy do najnizszej
klasy
Klasy zostaje wywolane pokolej
"""
class Klasa_odlegla(object):
    def __init__(self):
        print("Klasa odlegla ")

    def test5():
        print("Test odlegla")


class Klasa1(object):
    def __init__(self):
        print("Klasa 1")

    def test():
        print("Test 1")
```

```
class Klasa2(Klasa1):
    def __init__(self):
        Klasa1.__init__(self)
        print("Klasa 2")

    def test2():
        print("Test 2")


class Klasa3(Klasa2):
    def __init__(self):
        Klasa2.__init__(self)
        print("Klasa 3")

    def test3():
        print("Test 3")


class Klasa4(Klasa3):
    def __init__(self):
        Klasa3.__init__(self)
        print("Klasa 4")

    def test4():
        print("Test 4")


class Klasa5(Klasa4, Klasa_odlegla):
    def __init__(self):
        Klasa4.__init__(self)
        Klasa_odlegla.__init__(self)
        print("Klasa 5")
        Klasa_odlegla.test5()
        Klasa4.test4()
        Klasa1.test()

print("Od klasy 2 ")
k2 = Klasa2()
print("Od klasy 3 ")
k3 = Klasa3()
print("Od klasy 5")
k5 = Klasa5()
```

Wynik:

```
zad 3 =
Od klasy 2
Klasa 1
Klasa 2
Od klasy 3
Klasa 1
Klasa 2
Klasa 3
Od klasy 5
Klasa 1
Klasa 2
Klasa 3
Klasa 4
Klasa odlegla
Klasa 5
Test odlegla
Test 4
Test 1
```

Zad 4:

```
#W koljenosc wywoluje klasy zaczynając od 5
class Klasa_odlegla(object):
    def foo(self, numer_klasy):
        print("Klasa Klasa_odlegla wywolala Klase ", numer_klasy)

    def test5():
        print("Test odlegla")


class Klasa1(object):
    def foo(self, numer_klasy):
```

```
        print("Klasa 1 wywolala Klase ", numer_klasy)
        super().foo("1")

    def test():
        print("Test 1")


class Klasa2(Klasa1):
    def foo(self, numer_klasy):
        print("Klasa 2 wywolala Klase ", numer_klasy)
        super().foo("2")

    def test2():
        print("Test 2")


class Klasa3(Klasa2):
    def foo(self, numer_klasy):
        print("Klasa 3 wywolala Klase ", numer_klasy)
        super().foo("3")

    def test3():
        print("Test 3")


class Klasa4(Klasa3):
    def foo(self, numer_klasy):
        print("Klasa 4 wywolala Klase ", numer_klasy)
        super().foo("4")

    def test4():
        print("Test 4")

class Klasa5(Klasa4, Klasa_odlegla):
    def foo(self):
        print("Wywolanie klasy 5")
        super().foo("5")


k5 = Klasa5()
k5.foo()
print(Klasa5.__mro__)
```

Wynik:

```
zad 4 =

Wywolanie klasy 5

Klasa 4 wywolala Klase  5

Klasa 3 wywolala Klase  4

Klasa 2 wywolala Klase  3

Klasa 1 wywolala Klase  2

Klasa Klasa_odlegla wywolala Klase  1
```

Zad 5:

```
print("\nzad 5 =")
#dynamiczna klase utworzona za pomoca type()  o nazwie Klasa ktora ma
dwa atrybuty
Osoba = type('Osoba',(), dict(wiek=25, imie="Karol"))
osoba = Osoba()
print(osoba.__class__)
print(osoba.wiek)
print(osoba.imie)
```

Wynik:

```
zad 5 =

<class '__main__.Osoba'>

25

Karol
```