Klasy – tworzenie nowych typów danych

**Object-oriented programming**

https://en.wikipedia.org/wiki/Procedural\_programming

https://en.wikipedia.org/wiki/Object-oriented\_programming

https://en.wikipedia.org/wiki/Software\_design\_pattern

**INTRODUCTION**

**Obiekt – coś, co ma swój stan, atrubuty oraz swoje zachowania- metody**

Object-oriented programming (OOP) is a programming paradigm based on the concept of 'objects', which can contain data (attributes) and procedures (methods). Methods can access and often modify the attributes of the object with which they are associated (objects have a notion of "this" or "self"). Computer programs are designed by making them out of objects that interact with one another.

Classes sometimes correspond to things found in the real world.

**Inheritence** in typically used for code reuse. #dziedziczenie – jedna klasa uszczegóławia drugą

**Encapsulation** is an object-oriented programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Zamykanie w czarnej skrzynce pewnej funkcjonalności

Objects can contain other objects in their instance variables; this is known as object **composition**. #clasa w clasie

**A software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is a template for how to solve a problem that can be used in many different situations. #wzorce projektowe

Design patterns had originally been categorized into 3 sub-classifications based on kind of problem they solve.   
(a) Creational patterns (abstract factory, singleton)   
(b) Structural patterns (adapter, decorator)   
(c) Behavioural patterns (iterator, visitor)

**PYTHON**

# A simple example class.

class MyClass:

def speak(self): # a sample method

#self – dana instancja tej klasy przetwarzana wewnątrz metody

#metody wewnątrz klasy prawie zawsze mają self na początku

print("Hello")

# Driver code.

instance = MyClass()

instance.speak() # Hello

MyClass.speak(instance) # the same, instance as self, python zamienia zawsze na takie wywołanie

Data hiding (\_variable, \_\_hidden).

Pojedyncze podkreślenie przed nazwą zmiennej– prywatna zmienna

Podwójne podkreślenie – prywatna zmienna ze zmianą nazwy, szczególnie w klasach dziedziczonych żeby nie było kolizji

**Polynomials - definitions**

https://en.wikipedia.org/wiki/Polynomial

https://en.wikipedia.org/wiki/Horner%27s\_method

**INTRODUCTION**

A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponentiation of variables. Polynomials appear in many areas of mathematics and science.

A polynomial of a single variable x:

P(x) = x^2 - 4x + 7

The polynomial P defines a function

x → P(x),

where x can be a number or a polynomial.

Coefficients are usually numbers.

-4x is a (linear) term, -4 is a coefficient, the degree of x is one.

5 is a 'constant' polynomial (degree 0).

2x +3 is a 'linear' polynomial (degree 1).

3x^2 +5x -8 is a 'quadratic' polynomial (degree 2).

2x^3 -4x^2 +6x -1 is a 'cubic' polynomial (degree 3).

P(x) = p\_0 + p\_1 \* x + p\_2 \* x^2 = \sum\_i p\_i \* x^i, # LaTeX notation

Q(x) = q\_0 + q\_1 \* x + q\_2 \* x^2 = \sum\_i q\_i \* x^i.

Addition P + Q (the associative law of addition is used)

P(x) + Q(x) = (p\_0 + q\_0) + (p\_1 + q\_1) \* x + (p\_2 + q\_2) \* x^2.

P(x) + Q(x) = \sum\_i (p\_i + q\_i) \* x^i.

Substraction P - Q

P(x) - Q(x) = (p\_0 - q\_0) + (p\_1 - q\_1) \* x + (p\_2 - q\_2) \* x^2.

P(x) - Q(x) = \sum\_i (p\_i - q\_i) \* x^i.

Multiplication P \* Q (the distributive law is repeatedly applied)

P(x) \* Q(x) = p\_0 \* q\_0 + p\_0 \* (q\_1 \* x) + p\_0 \* (q\_2 \* x^2)

+ (p\_1 \* x) \* q\_0 + (p\_1 \* x) \* (q\_1 \* x) + (p\_1 \* x) \* (q\_2 \* x^2)

+ (p\_2 \* x^2) \* q\_0 + (p\_2 \* x^2) \* (q\_1 \* x) + (p\_2 \* x^2) \* (q\_2 \* x^2).

P(x) \* Q(x) = p\_0 \* q\_0

+ (p\_0 \* q\_1 + p\_1 \* q\_0) \* x

+ (p\_0 \* q\_2 + p\_1 \* q\_1 + p\_2 \* q\_0) \* x^2

+ (p\_1 \* q\_2 + p\_2 \* q\_1) \* x^3

+ (p\_2 \* q\_2) \* x^4.

P(x) \* Q(x) = \sum\_i \sum\_j p\_i \* q\_j \* x^{i+j}.

Composition P(Q)

P(Q(x)) = p\_0 + p\_1 \* Q(x) + p\_2 \* Q(x)^2.

**HORNER'S METHOD**

Horner's method (or Horner's scheme) is an algorithm for polynomial evaluation. It allows the evaluation of a polynomial of degree n with only n multiplications and n additions.

P(x) = p\_0 + p\_1 \* x + p\_2 \* x^2 + p\_3 \* x^3. # 6 mul, 3 add

P(x) = p\_0 + x \* (p\_1 + x \* (p\_2 + x \* (p\_3))). # 3 mul, 3 add, Horner

# First approach.

t = p\_3,

t = t \* x + p\_2,

t = t \* x + p\_1,

t = t \* x + p\_0.

# Second approach.

t = 0,

t = t \* x + p\_3,

t = t \* x + p\_2,

t = t \* x + p\_1,

t = t \* x + p\_0.

# Polynomials - interface

https://www.python-course.eu/polynomial\_class\_in\_python.php

### INTRODUCTION

Polynomials are implemented in many computer systems (Mathematica, Maple, Sage) and python modules (numpy, sympy). The interface of polynomials is not unique.

Poly – nazwa klasy stworzona przez nas

+-------------------------+-----------------------+

| Operation | Method |

+-------------------------+-----------------------+

| poly1 = Poly(c, n) | \_\_init\_\_, constructor |

| poly2 = Poly(c) | a constant polynomial |

| poly1 + poly2 | \_\_add\_\_ |

| poly1 - poly2 | \_\_sub\_\_ |

| poly1 \* poly2 | \_\_mul\_\_ |

| +poly1 | \_\_pos\_\_ |

| -poly1 | \_\_neg\_\_ |

| poly1 == poly2 | \_\_eq\_\_ |

| poly1 != poly2 | \_\_ne\_\_ |

| poly1.degree() | the degree of poly1 |

| poly1.is\_zero() | zero coefficients |

| len(poly1) | \_\_len\_\_, monomials |

| poly1(x) | \_\_call\_\_ |

| poly1.combine(poly2) | composition |

| poly1\*\*n, pow(poly1, n) | \_\_pow\_\_ |

| del poly1 | remove the name poly1 |

+-------------------------+-----------------------+

P(x) = p\_0 + p\_1 \* x + p\_2 \* x^2

will be created as

poly1 = Poly(p\_0) + Poly(p\_1, 1) + Poly(p\_2, 2)

**Polynomials - constructors**

https://docs.python.org/3/tutorial/classes.html

https://www.python-course.eu/polynomial\_class\_in\_python.php

**INTRODUCTION**

Creating a new class creates a new 'type' of object, allowing new 'instances' of that type to be made. A class in Python is defined by a 'class' statement (a compound statement). Class names should be in a StudlyCups style.

When a class definition is entered, a new namespace is created, and used as the local scope.

# Syntax.

class ClassName: # a 'class object' will be created; duże litery złączone

docstring # optional

statements

class DerivedClassName(BaseClassName): # inheritance; dziedziczenie

#Derived – nowa (potomek - child), bazowa (parent)– w nawiasie

docstring # optional

statements

class DerivedClassName(Base1, Base2, Base3): # multiple inheritance

docstring # optional

statements

# Now ClassName has a reference to the class object.

instance = ClassName() # 'calling' a class object

ClassName.\_\_doc\_\_ # access to the docstring

instance.\_\_class\_\_ # access to the class object

**POLYNOMIALS**

class Poly:

"""The class for polynomials of a single variable."""

def \_\_init\_\_(self, c=0, n=0): # a special method (constructor)

#self – zawsze pierwszy w danej metodzie

self.data = (n+1) \* [0]

self.data[-1] = c

def degree(self): # a method

return len(self.data) -1

def is\_zero(self): # a method, data can be [0], [0, 0], ...

return all(c == 0 for c in self.data)

# Testing the constructor.

poly1 = Poly(3, 5)

# Class 'instantiation', poly1 is a new 'instance'.

# \_\_init\_\_() is automatically invoked.

assert len(poly1.data) == 6 #nie wiem co to jest assert

assert poly1.data == [0, 0, 0, 0, 0, 3] # data is an instance attribute

assert poly1.degree() == 5 # calling a method

assert Poly.degree(poly1) == 5

assert not poly1.is\_zero()

assert Poly(0, 3).is\_zero() # data is [0, 0, 0, 0]

poly1.value = 124 # a new instance attribute

**CLASS ATTRIBUTES #atrybut klasy, taki sam dla wszystkich instancji**

Class variables (attributes) are shared by all instances of the class.

class Poly:

counter = 0 # counting created polynomials, a class attribute

def \_\_init\_\_(self, c=0, n=0): # a special method

self.data = (n+1) \* [0]

self.data[-1] = c

Poly.counter += 1 # using a class attribute

#Poly.counter = Poly.counter + 1 # the same

# C/C++: 'a (op)= b' is equal to 'a = a (op) b', (op) can be +,-,\*,/

def \_\_del\_\_(self):

# It is called when the reference counter drops to zero.

# Not exactly a destructor from C++.

Poly.counter -= 1

#Poly.counter = Poly.counter - 1 # the same

assert Poly.counter == 0

poly1 = Poly(3, 5)

assert Poly.counter == 1

assert poly1.counter == 1 # access from the instance (read-only)

del poly1

assert Poly.counter == 0

**STRING REPRESENTATIONS**

class Poly:

def \_\_str\_\_(self): # the informal string representation, str(instance)

return str(self.data)

# it can be also "p\_0 + p\_1 x + p\_2 x^2 + ..."

def \_\_repr\_\_(self): # the formal (canonical) string representation, repr(instance)

if self.is\_zero(): # calling a method inside the class, Poly.is\_zero(self)

return "Poly()"

else:

L = list()

for (i, c) in enumerate(self.data):

if c != 0:

L.append("Poly({}, {})".format(repr(c), i))

return " + ".join(L)

poly1 = Poly(3, 5)

poly1.data[0] = -2 # addition is not implemented yet ...

print(str(poly1)) # [-2, 0, 0, 0, 0, 3]

print(repr(poly1)) # Poly(-2, 0) + Poly(3, 5)

**Polynomials - operations**

https://docs.python.org/3/tutorial/classes.html

**COMPARISONS**

class Poly:

def \_\_eq\_\_(self, other): # poly1 == poly2

return (self - other).is\_zero() # it works when the substraction is defined

def \_\_ne\_\_(self, other): # poly1 != poly2

return not self == other

# There is a problem with <, <=, >, >=.

**ADDITION, SUBSTRACTION, MULTIPLICATION**

class Poly:

def \_\_add\_\_(self, other): # poly1 + poly2

new\_poly = Poly(0, max(len(self.data), len(other.data)) -1)

for (i, c) in enumerate(self.data):

new\_poly.data[i] += c

for (i, c) in enumerate(other.data):

new\_poly.data[i] += c

return new\_poly

def \_\_sub\_\_(self, other): # poly1 - poly2

new\_poly = Poly(0, max(len(self.data), len(other.data)) -1)

for (i, c) in enumerate(self.data):

new\_poly.data[i] += c

for (i, c) in enumerate(other.data):

new\_poly.data[i] -= c

return new\_poly

def \_\_mul\_\_(self, other): # poly1 \* poly2

new\_poly = Poly(0, len(self.data) + len(other.data) -2)

for (i, c) in enumerate(self.data):

for (j, d) in enumerate(other.data):

new\_poly.data[i+j] += c \* d

return new\_poly

def \_\_pos\_\_(self): # +poly1 = (+1)\*poly1

return self

def \_\_neg\_\_(self): # -poly1 = (-1)\*poly1

new\_poly = Poly()

new\_poly.data = [-c for c in self.data]

return new\_poly

poly1 = Poly(3, 5)

poly2 = Poly(1, 2)

assert poly1 == poly1 # testing comparisons

assert poly1 != poly2 # testing comparisons

assert poly1 \* poly2 == Poly(3, 7)

assert poly1.\_\_mul\_\_(poly2) == Poly(3, 7)

assert Poly.\_\_mul\_\_(poly1, poly2) == Poly(3, 7)

**COMPOSITION**

class Poly:

def \_\_call\_\_(self, x): # Horner, polynomials are callable now

result = 0

for c in reversed(self.data):

result = result \* x + c

return result

def combine(self, other): # Horner

new\_poly = Poly()

for c in reversed(self.data):

new\_poly = new\_poly \* other + Poly(c)

return new\_poly

poly1 = Poly(3, 5)

poly2 = Poly(1, 2)

assert poly1(2) == 96

assert poly2(9) == 81

assert poly1.combine(poly2) == Poly(3, 10) # 3\*(x^2)^5 = 3\*x^10

**EXERCISES**

Add new methods to the Poly class.

class Poly:

def copy(self): pass # shallow copy

def \_\_pow\_\_(self, n): pass # poly1 \*\* n, pow(poly1, n)

def \_\_len\_\_(self): pass # len(poly1), the number of monomials

def diff(self): pass # derivative

def integrate(self): pass # indefinite integral

**Special methods**

https://docs.python.org/3/reference/datamodel.html

zaczynają się I kończą dwoma podkreślnikami

**INTRODUCTION**

A class can implement certain operations that are invoked by special syntax by defining methods with special names. This is Python’s approach to 'operator overloading', allowing classes to define their own behavior with respect to language operators.

\_\_new\_\_ called to create a new instance of class

\_\_init\_\_ called after the instance has been created (a constructor)

\_\_del\_\_ Called when the instance is about to be destroyed (a finalizer, a destructor).

Note that 'del x' decrements the reference count for x by one,

and 'x.\_\_del\_\_()' is only called when x’s reference count reaches zero.

\_\_str\_\_ Called by str() and the built-in functions format()

and print() to compute the 'informal' string representation of an object.

The return value must be a string object.

\_\_repr\_\_ called by repr(), must return a string

\_\_format\_\_ called by format() to produce a 'formatted' string representation

\_\_bytes\_\_ called by bytes() to compute a byte-string representation

\_\_hash\_\_ called by hash(), should return an integer

If x == y then always hash(x) == hash(y).

\_\_cmp\_\_ called by cmp() in Py2, should return -1 or 0 or 1

cmp(1, 4) returns -1

cmp(3, 3) returns 0

cmp(5, 3) returns 1

We can define our cmp() function in Py3:

cmp = lambda x, y: (x > y) - (x < y)

Rich comparison methods. #zwraca true jeśli porównanie jest prawdziwe

\_\_lt\_\_ called by x < y

\_\_gt\_\_ called by x > y

\_\_eq\_\_ called by x == y

\_\_le\_\_ called by x <= y

\_\_ge\_\_ called by x >= y

\_\_ne\_\_ called by x != y

assert (x < y) == (y > x) # may be used by Python

assert (x <= y) == (y >= x) # may be used by Python

assert (x == y) == not (x != y) # not always True!

PEP 207

min() and list.sort() operations only use the < operator (less).

max() only use the > operator.

'in' and 'not in' operators and dictionary lookup only use the == operator.

The reflexivity rules are assumed by Python.

\_\_bool\_\_ called by bool() to implement truth value testing, should return False or True

\_\_nonzero\_\_ truth value testing in Py2

\_\_bool\_\_ = \_\_nonzero\_\_ compatibility

Emulating container types.

\_\_len\_\_ called by len(), should return the length of the container;

if len(x) == 0 and \_\_bool\_\_ is not defined then x is considered False

\_\_getitem\_\_(self, key) called to implement evaluation of 'X[key]'

\_\_setitem\_\_(self, key, value) called to implement assignment 'X[key] = value'

\_\_delitem\_\_(self, key) called to implement deletion 'del X[key]'

\_\_iter\_\_ called when an iterator is required for a container

\_\_reversed\_\_ called by reversed() to implement reverse iteration

\_\_contains\_\_(self, item) called to implement membership test operators

'item in X' and 'item not in X'

\_\_copy\_\_ shallow copy, Y = copy.copy(X)

\_\_deepcopy\_\_ deep copy, Y = copy.deepcopy(X, memo)

Emulating numeric types.

\_\_add\_\_ called by x + y

\_\_sub\_\_ called by x - y

\_\_mul\_\_ called by x \* y

\_\_mod\_\_ called by x % y (remainder)

\_\_divmod\_\_ called by divmod(x, y), return (a // b, a % b) for integers

\_\_pow\_\_ called by pow(x, y) and x \*\* y

\_\_floordiv\_\_ called by x // y (quotient)

\_\_truediv\_\_ called by x / y (Py3)

\_\_div\_\_ called by x / y (Py2)

\_\_div\_\_ = \_\_truediv\_\_ compatibility (not always good)

\_\_pos\_\_ called by +x

\_\_neg\_\_ called by -x

\_\_abs\_\_ called by abs(x)

\_\_invert\_\_ called by ~x

\_\_complex\_\_ called by complex(x)

\_\_int\_\_ called by int(x)

\_\_long\_\_ called by long(x) (Py2)

\_\_float\_\_ called by float(x)

**Using collections**

https://docs.python.org/3/library/collections.html

**INTRODUCTION**

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, 'dict', 'list', 'set', and 'tuple'.

* namedtuple() - factory function for creating tuple subclasses with named fields
* deque - list-like container with fast appends and pops on either end
* Counter - dict subclass for counting hashable objects
* OrderedDict - dict subclass that remembers the order entries were added
* defaultdict - dict subclass that calls a factory function to supply missing values

**DEQUE**

#spoko kolekcja, prawie jak lista ale można dołączać tu i tam

import collections

# collections.deque([iterable[, maxlen]])

# Methods: copy(), clear(), count(), index(), insert()

# remove(), reverse(), rotate(n=1),

# extend(iterable), extendleft(iterable),

# append(), appendleft(),

# pop(), popleft().

d = collections.deque('ghi') # make a new deque with three items

print(d) # deque(['g', 'h', 'i'])

d.append('j') # add a new entry to the right side

d.appendleft('f') # add a new entry to the left side

print(d) # deque(['f', 'g', 'h', 'i', 'j'])

item1 = d.pop() # 'j', return and remove the rightmost item

item2 = d.popleft() # 'f', return and remove the leftmost item

list(d) # ['g', 'h', 'i'], list the contents of the deque

assert d[0] == 'g' and d[-1] == 'i' # fast only for the ends

'h' in d # True

for item in d: # iteration

print(item)

**Using timeit**

https://docs.python.org/3/library/timeit.html

**INTRODUCTION**

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one.

import timeit

# class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>, globals=None)

t1 = timeit.Timer(stmt="t=a ; a=b ; b=t", setup="a=1 ; b=2")

t2 = timeit.Timer(stmt="(a,b) = (b,a)", setup="a=1 ; b=2")

print(t1.timeit(number=10)) # number=1000000 is the default

print(t2.repeat(repeat=3, number=10)) # return a list with 3 results

print(min(t2.repeat(10))) # this is what we typically want

import random

import timeit

def mysort(L):

L[:] = sorted(L)

alist = list(range(pow(10,6)))

random.shuffle(alist)

t1 = timeit.Timer(stmt="mysort(alist)",

setup="from \_\_main\_\_ import mysort, alist") # using strings

#t1 = timeit.Timer(lambda: mysort(alist)) # using a callable

print(t1.timeit(1)) # single run

# Three functions are available.

#

# timeit.timeit(stmt='pass', setup='pass', timer=<default timer>,

# number=1000000, globals=None)

#

# timeit.repeat(stmt='pass', setup='pass', timer=<default timer>,

# repeat=5, number=1000000, globals=None)

#

# timeit.default\_timer()

# The default timer, which is always time.perf\_counter() [Py3.3+].

print(timeit.timeit("sum(range(pow(10,6)))", number=1))

print(timeit.timeit(lambda: sum(range(pow(10,6))), number=1))

print(timeit.repeat("sum(range(pow(10,6)))", repeat=3, number=1))

print(timeit.repeat(lambda: sum(range(pow(10,6))), repeat=3, number=1))

**Homework 6**

Send your homework as a single ZIP archive with your files, or send a link to your GitHub repo. Try to keep line lengths below 80 characters.

**EXERCISE 6.1**

Create 3D vectors as a Python class.

class Vector:

def \_\_init\_\_(self, x, y, z):

self.x = x

self.y = y

self.z = z

def \_\_repr\_\_(self): pass # return string "Vector(x, y, z)"

def \_\_eq\_\_(self, other): pass # v == w

def \_\_ne\_\_(self, other): # v != w

return not self == other

def \_\_add\_\_(self, other): pass # v + w

# Hint: return Vector(...)suma x, xuma y, suma z

def \_\_sub\_\_(self, other): pass # v - w

def \_\_mul\_\_(self, other): pass # return the dot product (number)

def cross(self, other): pass # return the cross product (Vector)

def length(self): pass # the length of the vector

def \_\_hash\_\_(self): # we assume that vectors are immutable

return hash((self.x, self.y, self.z)) # recommended ?? co to hash

# Exemplary tests. Change values in your tests.

import math

v = Vector(1, 2, 3)

w = Vector(2, -3, 2)

assert v != w

assert v + w == Vector(3, -1, 5)

assert v - w == Vector(-1, 5, 1)

assert v \* w == 2

assert v.cross(w) == Vector(13, 4, -7)

assert v.length() == math.sqrt(14)

S = set([v, v, w])

assert len(S) == 2

print("Tests passed")

# two vectors are given (a vector is a 3-tuple in this reminder)

# a = (a1, a2, a3)

# b = (b1, b2, b3)

# the dot product (the result is a number)

# a · b = a1 \* b1 + a2 \* b2 + a3 \* b3

# the cross product (the result is a vector)

# a × b = (a2 \* b3 - a3 \* b2, a3 \* b1 - a1 \* b3, a1 \* b2 - a2 \* b1)