Lab6 Programowanie Zorientowane Obiektowo

Programując obiektowo tworzymy klasy – swojego rodzaju przepis na obiekt.
Obiekty są instancją, egzemplarzami klasy – są stworzone na bazie klasy. Tworząc klasę definiujemy ogólne zachowanie danej kategorii obiektów, innymi słowy tworząc obiekt na bazie klasy nadajemy mu ogólne zachowanie.

Klasa jest definicją typu obejmującego pewną grupę danych, oraz funkcje na nich operujące.
Klasa opisuje co dany egzemplarz posiada (pola) i jak się zachowuje (metody).

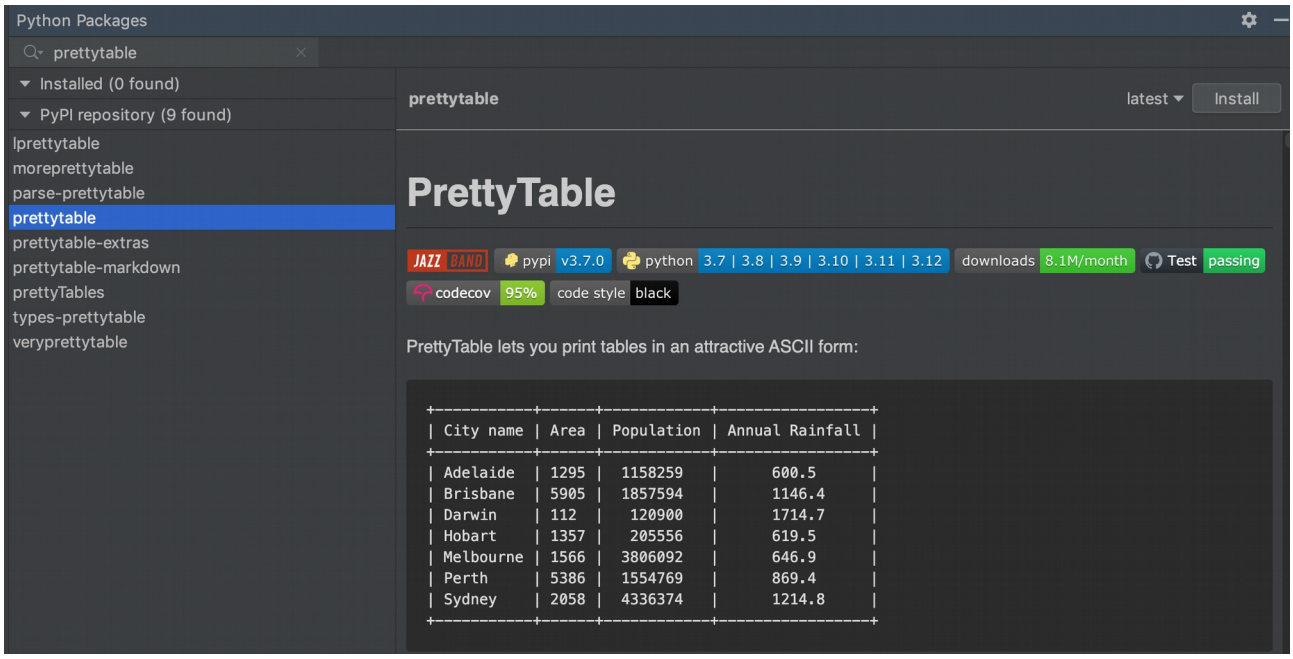
W Pythonie można wykorzystywać klasy udostępnione przez innych programistów.

Za pomocą klas można modelować praktycznie wszystko.

W Poprzednim zadaniu należało stworzyć za pomocą słownika spis studentów. Zaprojektowanie poprawnego wyświetlania w konsoli danych dotyczących studentów w postaci odpowiednio sformatowanej tabeli może być czasochłonnym zadaniem. Możemy posłużyć się gotową klasą przygotowującą taką tabelkę.

Istnieje wiele modułów rozwiązujących to zadanie, przykładowym jest PrettyTable.
PrettyTable możemy zainstalować za pomocą pip, pip jest to prosty system zarządzania pakietami używany do instalacji i zarządzania pakietami, które można znaleźć w Indeks pakietów Pythona (PyPi).

https://ptable.readthedocs.io/en/latest/tutorial.html




```
#
from prettytable import PrettyTable
x = PrettyTable()
```

Funkcja PrettyTable() tworzy obiekt Klasy PrettyTable – funkcja nazywa się tak samo jak nazwa klasy, której instancję tworzymy.

Przykładowo lista słowników:

```
student = [{"email": "student1@gmail.com",
           "nazwisko": "Nowak"},
            {"email": "student2@gmail.com",
           "nazwisko": "Kowalski"}
           ]
```

Ustawienie nazw kolumn obiektu PrettyTable:

```
x.field_names = ["email","nazwisko"]
```

dodanie elementów do tabeli za pomocą metody obiektu klasy PrettyTable:

```
x.add_row([email,nazwisko])
```

Pobranie tych danych ze słownika i dodanie może wyglądać np. tak:

```
for s in student:
    email = ""
    nazwisko = ""
    for k,v in s.items():
        if k == "email":
            email = v
        elif k == "nazwisko":
            nazwisko = v
    x.add_row([email,nazwisko])
```

Wyświetlenie:

```
print(x)
```

Jak możemy sami stworzyć własną klasę i egzemplarze takiej klasy ?

Utwórzmy klasę reprezentującą prostą strukturę danych z dostępem ograniczonym – Stos.
Stos jest strukturą, w której elementy dodawane są na jego koniec (zwany "wierzchołkiem") i z końca są ściągane.

Ten typ danych musi wspierać następujące operacje:
Stack() - tworzenie nowego pustego stosu
push(item) wstawienie nowego elementu na wierzchołek stosu
pop() pobranie (usunięcie z jednoczesnym zwróceniem wartości) elementu z wierzchołka stosu
peek() odczytuje wartość z wierzchołka stosu. Nie usuwa jej (stos pozostaje niezmieniony)
isEmpty() testuje, czy stos jest pusty
size() liczba elementów znajdujących się na stosie

```
class Stack:
    def __init__(self,max_size):
        self._stos = list()
        self._max_size = max_size
```

Do stworzenia klasy używamy słowa kluczowego class oraz nazwy klasy.
Podczas tworzenia nowego obiektu wywoływana jest automatycznie specjalna metoda __init__().

Metoda __init__ nazywana jest często konstruktorem, poprawna nazwa to metoda inicjalizacyjna. Wywoływana jest ona jako pierwsza podczas inicjalizacji nowego obiektu, tuż po jego utworzeniu. Służy do ustawienia pól nowego obiektu.

Parametr self jest niezbędny – jest to odwołanie do danego egzemplarza klasy. Metody przyjmują jako pierwszy parametr obiekt, na rzecz którego są wywoływane.

Napiszmy dwie metody realizujące funkcjonalność dodawania i usuwania elementów:

```
def push(self,e):
    if len(self._stos) < self._max_size:
        self._stos.append(e)
    else:
        raise Exception("Stack overflow")
def pop(self):
    if len(self._stos) > 0:
        return self._stos.pop()
    else:
        raise Exception("pop from empty stack")
```

gdy stos pełny i chcemy dodać element, lub pusty i chcemy zdjąć element zostanie wyrzucony wyjątek.

Aby wyświetlić zawartość stosu za pomocą funkcji print należy napisać metodę __str__, wywoływana jest gdy używamy funkcji str() i również przy print(), gdy jej nie napiszemy efekt użycia metody print może wyglądać podobnie jak: <Stack.Stack object at 0x1103cd890>

```
def __str__(self):
    return str(self._stos)
```

Całość wygląda następująco:

```
class Stack:
    def __init__(self,max_size):
        self._stos = list()
        self._max_size = max_size
    def push(self,e):
        if len(self._stos) < self._max_size:
            self._stos.append(e)
        else:
            raise Exception("Stack overflow")
    def pop(self):
        if len(self._stos) > 0:
            return self._stos.pop()
        else:
            raise Exception("pop from empty stack")
    def __str__(self):
        return str(self._stos)
```

Możemy teraz zaimportować nasz moduł i używać naszej klasy:

```
import Stack
```

```
stos = Stack.Stack(2)
stos.push("element1")
stos.push("element2")
#stos.push("element3")
print(stos)
print(stos.pop())
print(stos.pop())
#print(stos.pop())
print(stos)
```

W Pythonie jest jedna metoda inicjalizacyjna. Można natomiast używać wartości domyślnych:

```
    def __init__(self,max_size=10):
        self._stos = list()
        self._max_size = max_size
```

```
stos = Stack.Stack()
stos.push("element1")
stos.push("element2")
stos.push("element3")
print(stos)
```

W większości obiektowych języków programowania występuje pojęcie kontroli dostępu.
Niektóre atrybuty i metody obiektu mogą zostać określone jako prywatne, co oznacza, że jedynie dany obiekt będzie miał do nich dostęp.
Inne są oznaczane jako chronione ,w takim przypadku jedynie dana klasa oraz jej klasy pochodne będą mieć do nich dostęp. Wszystkie pozostałe atrybuty i metody są uznawane za publiczne: wszystkie inne obiekty będą mogły uzyskać do nich dostęp.
W Pythonie wszystkie metody i atrybuty klas są publicznie dostępne. Zgodnie z konwencją nazwy wszystkich metod i atrybutów przeznaczonych do użytku wewnętrznego są poprzedzane znakiem podkreślenia (_). Podwójne podkreślenie oznacza atrybuty i funkcje pseudo prywatne, jednak nadal można się do nich odwołać:

```
stos = Stack.Stack(2)
stos.push("element1")
stos.push("element2")
stos._Stack__stos.pop()
print("stos:",stos._Stack__stos)
```

Klasa student reprezentująca studenta oraz jego oceny może wyglądać następująco

```
class Student:
    def __init__(self, email, name, surname):
```

```
        self.all_grade = {
            "project": -1,
            "l_1": -1,
            "l_2": -1,
            "l_3": -1,
            "h_1": -1,
            "h_2": -1,
            "h_3": -1,
            "h_4": -1,
            "h_5": -1,
            "h_6": -1,
            "h_7": -1,
            "h_8": -1,
            "h_9": -1,
            "h_10": -1,
            "grade": -1,
        }
        self.email = email
        self.name = name
        self.surname = surname
        self.status = None
```

Specjalne nazwy metod:
Klasy mogą implementować pewne operacje, które możemy następnie wywoływać przy użyciu specjalnej składni (np. używając operatorów arytmetycznych, czy porównań. Do ich definicji używa się metod o specjalnych nazwach.
https://pl.python.org/docs/ref/node15.html

Aby móc wyświetlić za pomocą print nasz obiekt musimy napisać metodę __str__:

```
def __str__(self):
    return str(self.email + " " +
               self.name + " " +
               self.surname + " " +
               str(self.all_grade) + " " +
               str(self.status)
```

Aby móc wyświetlać obiekt naszej klasy umieszczony w słownikach:

```
def __repr__(self):
    return self.__str__()
```

Jeżeli chcielibyśmy porównywać dwa obiekty w konkretny sposób należy napisać metodę __eq__, np.:

```
def __eq__(self, other):
    if not isinstance(other, Student):
        return NotImplemented
    return self.email == other.email
```

Metody statyczne:
Możemy napisać metody wywoływane na klasie a nie na instancji, np.:

```
@staticmethod
```

```
def compareStudentSurname(x,y):
    return x.surname >= y.surname
```

Czy nazwisko pierwszego studenta jest „większe-równe” (alfabetycznie) niż drugiego:

```
#test:
import Student
student = Student.Student("lukasz@gmail.com","lukasz","kwasniewicz")
student2 = Student.Student("aadamski@gmail.com","adam","adamski")
print(Student.Student.compareStudentSurname(student,student2))
```

Funkcje w pythonie mogą zwracać wiele wartości:

```
def getPersonalData(self):
    return self.email, self.name, self.surname
```

```
#test:
email, name, surname = student.getPersonalData()
print(email, name, surname)
```

Funkcje jako parametr:
W pythonie możemy przekazywać funkcję jako parametr:
np. sprawdzenie kolejności alfabetycznej 2 studentów (wartość kodu liter) po nazwisku lub po imieniu mogłaby mieć postać:

```
@staticmethod
def compareStudentSurname(x,y):
    return x.surname >= y.surname
@staticmethod
def compareStudentName(x,y):
    return x.name >= y.name
```

w teście:

```
def alphabeticGrater(x, y, func):
    return func(x, y)
```

```
student = Student.Student("lukasz@gmail.com", "lukasz", "kwasniewicz")
student2 = Student.Student("aadamski@gmail.com", "zbigniew", "adamski")
```

```
print(alphabeticGrater(student, student2,Student.Student.compareStudentSurname))
print(alphabeticGrater(student, student2,Student.Student.compareStudentName))
```

Zadania:

Zadanie1
Napisz klasę **MyLinkedList** – która będzie posortowana,wiązana oraz jednostronna.
Używając dodatkowo klasy:

```
class Element:
    def __init__(self, data=None, nextE=None):
        self.data = data
        self.nextE = nextE
```

Klasa MyLinkedList zawiera pola:

```
  ...
    self.head = None
    self.tail = None
    self.size = 0
```

z metodami:
__str__ - reprezentacja napisowa listy – wszystkie elementy listy (5%)
get(self, e) – zwraca element, (5%)
delete(self,e) – usuwa wskazany element,
(25%)
append (self, e, func=None) – dodaje elementy do listy w sposób posortowany. (25%)
func – jaki będzie warunek sortownia – określi funkcja, jeżeli None – zwykłe porównanie 2 obiektów za pomocą >=

Zadanie2 (40%):
Zmodyfikować zadanie z poprzednich zajęć tak aby używała klasy Student ,MySortedList oraz aby ocenianie przebiegało zgodnie z zasadami zaliczenia:
1 ocena za projekt – 40 pkt
3 oceny z list z zadaniami – 20 pkt każda
Oceny z prac domowych.
W zależności od średniej z prac domowych, należy zastąpić najsłabsze oceny z list od 1 do 3.
60% - jedna lista (20pkt)
70% - dwie listy (40pkt)
80% - trzy listy (60 pkt)

Ocenę końcową można wystawić tylko kiedy wszystkie oceny cząstkowe są wystawione.

Należy umożliwić przy wysyłaniu emaila podanie nowego statusu, oraz umożliwić podanie statusów, przy których chcemy wysyłać email. E-mail można wysłać zawsze jeżeli status na to pozwala.

Przykładowy plik: (-1 oznacza, że ocena nie została jeszcze wystawiona)

```
none@gmail.com,ann,none,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,None
test@gmail.com,alice,novak,40,20,20,20,100,100,100,100,100,100,100,100,100,100,5
,MAILED_FINAL
test2@gmail.com,bob,bobowsky,40,20,20,20,100,100,100,100,100,100,100,100,100,100
,5,GRADED
test3@gmail.com,chris,czajkowsky,-1,20,20,-
1,100,100,100,100,100,100,100,100,100,-1,-1,HOME_9
test4@gmail.com,Derek,Derkowsky,-1,20,20,-
1,100,100,100,100,100,100,100,100,100,-1,-1,HOME_9_MAILED
```