Klasy i programowanie obiektowe

Python wspiera metodykę programowania obiektowego, chociaż jej nie wymusza (jak C#/Java) – w tym jest podobny do starszych języków programowania (jak Pascal, C, C++); Podobnie jest też w przypadku innych języków skryptowych (PHP, Javascript). Możliwości programowania obiektowego w samym szkielecie Pythona dość ograniczone – np. nie ma interfejsów, klas abstrakcyjnych, specyfikatorów dostępu, … – ale niektóre funkcjonalności można dodać przez import modułów i metaprogramowanie, np. dekoratory.
Python jest językiem zorientowanym na przetwarzanie kolekcji (spuścizna języków SETL i ABC), co jest zauważalne także w wielu aspektach programowania obiektowego.
Niektóre aspekty programowania obiektowego są bardziej konwencją, niż skutkiem użycia konstrukcji składniowych, np. "magiczne funkcje", co również jest dość typowe dla języków skryptowych.

Klasy

Do definiowania funkcji służy słowo kluczowe (class). Obiekt jest tworzony przez "wywołanie" klasy jak funkcji

```
class Student:
    pass

s = Student()

print(s)  # <__main__.Student object at 0x02DAF940>
```

Metody

Metody definiuje się podobnie jak zwykłe funkcje, z jedną różnicą: każda metoda musi mieć jako pierwszy, obowiązkowy argument, instancję obiektu, zwyczajowo "self". Przy wywołaniu metody tego argumentu się nie podaje (zostaje dodany niejawnie). Operator dostępu ma postać kropki, ".".

```
class Student:
    def Print(self):
        print("<obiekt klasy Student>")

s = Student()
s.Print()
```

Inicjalizacja obiektów

Python formalnie nie ma konstruktorów klas (tworzenie obiektów w dokumentacji Pythona nosi nazwę instantiation), natomiast można zdefiniować inicjalizator – jest to metoda specjalna, o nazwie typu "dunder" `__init__`. Inicjalizator jest niejawnie wywoływany podczas tworzenia obiektu, zatem w praktyce spełnia funkcję konstruktora. Argument "self" jest wymagany i reprezentuje tworzony obiekt, któremu Python już przydzielił niezbędne zasoby. Poprzez atrybut self można dodawać atrybuty obiektu

```
class Student:
    def __init__(self):
        print("Tworzenie obiektu klasy Student…")

s = Student()   # niejawnie wywołane __init__
```

Atrybuty klasy

Atrybuty klasy można definiować w obrębie definicji klasy (muszą mieć nadaną wartość – ponieważ zmiennych się nie deklaruje). Są wspólne dla wszystkich obiektów, ale pod warunkiem, że są używane tylko do odczytu (modyfikacja tylko przez klasę)

```
class Student:
    uczelnia = 'AM'

    def gdzieStudiuje(self):
        print(self.uczelnia)

s1 = Student()
s1.gdzieStudiuje()  # 'AM'
Student.uczelnia = 'UMG'
s1.gdzieStudiuje()  # 'UMG'
```

Modyfikacja atrybutu klasy przez obiekt powoduje "przesłonięcie" atrybutu klasy przez atrybut obiektu (ale nie wpływa to na pozostałe obiekty tej klasy)

```
class Student:
    uczelnia = 'UMG'

s1 = Student()
s2 = Student()

s1.uczelnia = 'PG'
s1.uczelnia        # 'PG'

s2.uczelnia        # 'UMG'
```

Modyfikacja atrybutu klasy przez obiekt jest możliwa, ale trzeba odwołać się do niego przez nazwę klasy albo przez specjalny atrybut "__class__", który daje dostęp do wszystkich atrybutów klasy:

```
class Student:
    uczelnia = 'UMG'
    adres = 'Gdynia'
    def włam(self):
        self.__class__.uczelnia = 'PG'
        Student.adres = 'Gdańsk'


s1 = Student()
s1.włam()

s1.uczelnia        # 'PG'
```

Atrybuty obiektów są dynamiczne – można je dodawać podczas inicjalizacji albo później; można to zrobić w każdej metodzie klasy, przez argument self albo bezpośrednio przez obiekt. Dostęp do atrybutu obiektu z metod obiektu: zawsze przez self (inaczej jest to odwołanie do zmiennej spoza klasy!)

```
class Student:
    def __init__(self, album):
        self.album = album

    def print(self):
```

```
        print("album:", self.album)

s1 = Student(39765)
s1.album   # 39765
s1.print() # 'album: 39765'
```

Aby zachować jako taki porządek, należy wszystkie potrzebne atrybuty obiektu zdefiniować (przez nadanie im wartości) w inicjalizatorze; Można wykorzystać argumenty domyślne inicjalizatora, można też zainicjować atrybuty nie przekazane w wywołaniu

```
class Student:

    def __init__(self, album, rok = 1):
        self.album = album
        self.rok = rok
        self.oceny = {}

s = Student(39765) # rok = 1, oceny = {}
```

Metody "statyczne"

Python nie ma specyfikatorów dostępu, za to ma tzw. dekoratory, które m.in. umożliwiają definiowanie metod statycznych. Rozróżnia się przy tym metody klasy, `@classmethod`, oraz metody statyczne, `@staticmethod`

Metody klasy mogą być wywołane bezpośrednio przez klasę albo przez obiekt. Otrzymują jako pierwszy argument klasę (zwyczajowa nazwa "cls")

```
class Student:
    uczelnia = 'AM'

    @classmethod
    def nowaNazwa(cls, nazwa):
        cls.uczelnia = nazwa

Student.nowaNazwa('UMG')

s1 = Student()
s1.nowaNazwa('UMG')
```

Metody statyczne mogą być wywołane bezpośrednio przez klasę albo przez obiekt. Metody te NIE otrzymują klasy jako argumentu, są jak zwykłe funkcje, tylko "podpięte" pod klasę

```
class Student:
    uczelnia = 'AM'

    @staticmethod
    def nowaNazwa(nazwa):
        Student.uczelnia = nazwa

Student.nowaNazwa('UMG')

s1 = Student()
s1.nowaNazwa('UMG')
```

Metody statyczne najczęściej służą jako alternatywne konstruktory

```
class Student:

    @staticmethod
    def czytajDaneZBazyDanych(id):
        // odczytanie danych i utworzenie obiektu
        // … student = Student(…, …, …)
        return student

s1 = Student.czytajDaneZBazyDanych(idStudenta)
```

Programowanie obiektowe

Są trzy fundamentalne właściwości OOP: hermetyzacja (enkapsulacja), dziedziczenie oraz polimorfizm (wielopostaciowość), przy czym polimorfizm może być wspierany przez metody wirtualne, klasy abstrakcyjne oraz przez interfejsy. Spośród nich tylko dziedziczenie jest w Pythonie realizowane jak w innych językach obiektowych – enkapsulacja jest celowo słaba ("niewymuszona"), zaś polimorfizm opiera się na "kaczym typowaniu" (duck-typing). Klasy abstrakcyjne i interfejsy nie są obecne w szkielecie Pythona, ale mogą być realizowane przez dekoratory, po zaimportowaniu odpowiednich modułów

Metody i atrybuty prywatne i chronione

Python nie ma specyfikatorów dostępu (np. private, public, internal), zamiast tego elementy składowe klasy, których nazwa zaczynają się od "__" (ale nie są "dunder"), są uznawane za prywatne – nie są widoczne z zewnątrz obiektu oraz w klasach potomnych.
Elementy składowe klasy, których nazwa zaczyna się od "_", są przez uznawane za chronione – ale tylko nieoficjalnie

```
class Student:
    def __init__(self, email):
        self.__email = email

s1 = Student('student39765@umg.edu.pl')

s1.__email  # AttributeError: 'Student' object
            # has no attribute '__email'
```

Elementy "prywatne" tak naprawdę nie są prywatne – tylko ukryte; Dostęp do nich jest możliwy przez atrybut utworzony w ramach tzw. dekorowania nazw ("_" + klasa + atrybut):

```
class Student:
    def __init__(self, email):
        self.__email = email

s1 = Student('student39765@umg.edu.pl')

s1.__email          # błąd!
s1._Student__email  # ok, 'student39765@umg.edu.pl'
```

Na jednym z forów Pythona pojawiło się stwierdzenie (cytat z pamięci): Python nie wymusza prywatności. Woli, żebyś nie przekraczał progu, ponieważ nie zostałeś zaproszony, a nie dlatego, że ma strzelbę

Właściwości

Właściwość w programowaniu obiektowym oznacza atrybut klasy używany jak pole, ale wykonujący w tle przetwarzanie danych jak metoda – jest to połączenie funkcjonalności gettera i settera języka Java, ale pod wspólną nazwą
Właściwości można definiować przynajmniej na 3 sposoby:

- dekorator @property,
- wbudowaną funkcję property
- funkcje "magiczne", m.in. __getattr__ i __setattr__

W każdym przypadku można zdefiniować właściwość do odczytu i zapisu (tj. mającą getter i setter) albo tylko do odczytu (tylko getter)

Definicja właściwości przez dekorator @property

```
class Student:
    def __init__(self):
        self.ects = 0

    @property
    def DługEcts(self):
        return self.ects

    @DługEcts.setter
    def DługEcts(self, value):
        if value>=0:
            self.ects = value
        else
            raise ValueError("Dług nie może być ujemny")
```

Definicja właściwości przez funkcję property

```
class Student:
    def __init__(self):
        self.ects = 0

    def getEcts(self):
        return self.ects
    def setEcts(self, value):
        self.ects = value

    Ects = property(getEcts, setEcts)

s = Student()
s.Ects = 3      # s.setEcts(3)
print(s.Ects)   # s.getEcts()
```

Dziedziczenie

Python ma dziedziczenie po wielu klasach. Nazwy klas bazowych wpisuje się po nazwie definiowanej klasy, w nawiasach i oddzielone przecinkami.
Wszystkie metody klasy bazowej (oprócz "prywatnych") są dostępne dla klasy potomnej; Jeżeli klasa potomna definiuje metody obecne w klasie bazowej, to są one nadpisywane; Nie ma specjalnego oznaczania metod wirtualnych – wszystkie metody są wirtualne. Inicjalizator klasy potomnej może jawnie wywołać inicjalizator klasy bazowej

```
class Student (Osoba):

    def __init__():
```

```
Osoba.__init__()
```

Polimorfizm

Python nie ma interfejsów ani klas abstrakcyjnych (które w starszych językach, np. C/C++, zastępowały interfejsy). Wszystkie nowe języki – również skryptowe, np. PHP – mają interfejsy, a wiele tzw. dobrych praktyk opiera się na wykorzystaniu interfejsów (np. wzorce projektowe Gammy, "wstrzykiwanie interfejsów" w językach ze ścisłym typowaniem itd.). To nie znaczy, że coś podobnego nie działa w Pythonie – zamiast tego obowiązuje tzw. "kacze typowanie". Określenie to pochodzi z dyskusji o dziedziczeniu i relacji "jest" na forum Pythona; Kacze typowanie opiera się na konstatacji, że w Pythonie nie deklaruje się typu zmiennych – typ mają dane, a nie zmienne – zatem np. wywołanie metody się uda, jeżeli obiekt ją ma, a deklarowanie interfejsu w tej sytuacji jest zbędne, a przy okazji upraszcza kod, co jest cenne z punktu widzenia filozofii Pythona.

Dla przykładu, wywołanie funkcji ZapiszOrazDrukuj poniżej powiedzie się, jeżeli obiekt o nazwie osoba, należący do klasy Uczeń albo do klasy Student, ma metody Zapisz oraz Drukuj – ale to okaże się dopiero podczas wykonywania programu.

```
def ZapiszOrazDrukuj(obiekt):
    obiekt.Zapisz()
    obiekt.Drukuj()

if <jakiś warunek>:
    osoba = Uczeń()
else:
    osoba = Student()

ZapiszOrazDrukuj(osoba)
```

Funkcje specjalne

Funkcje specjalne (nazywane czasami magicznymi) mają nazwy typu "dunder". Wiele funkcji należących do szkieletu Pythona wywołuje odpowiednie funkcje specjalne – np. len(x) jest realizowane przez interpreter Pythona jako x.__len__(). Większość funkcji specjalnych można wywołać jawnie (np. len(x) albo x.__len__()), ale są wyjątki, np. __init__ jest wywoływana niejawnie w trakcie tworzenia obiektu, a jej jawne wywołanie powoduje rzucenie wyjątku programowego.

Aby funkcje specjalne mogły spełniać swoją rolę, muszą mieć określoną liczbę argumentów i właściwie je interpretować, np. __str__(self), __getitem__(self, key), __add__(self, other). W niektórych przypadkach istnieje więcej niż jeden sposób implementacji określonej funkcjonalności – interpreter próbuje użyć metody preferowanej, a jeżeli to się nie udaje, to metody alternatywnej; Np. iterator można zaimplementować przynajmniej na 3 sposoby.

Wyposażenie klasy w odpowiedni zestaw funkcji specjalnych umożliwia traktowanie obiektów tej klasy podobnie, jak typów wbudowanych, z użyciem operatorów, wyrażeń listowych, indeksowania itd., np.:

```
StudenciUMG = Students.ReadFromFile()

for student in StudenciUMG:
    print(student)
```

```
Kandydaci = Rekrutacja.ReadFromFile()

Przyjęci = (k for k in Kandydaci if k.punkty >= limit)
for kandydat in Przyjęci:
    StudenciUMG += kandydat
```

W dalszej części tego punktu przedstawiono wskazówki implementacji kilku wybranych funkcji specjalnych. Więcej informacji o funkcjach specjalnych można znaleźć w poradnikach internetowych, np.:

- Python.org – kompletna specyfikacja funkcji specjalnych wraz z opisem ich przeznaczenia i sposobu wykorzystania przez szkielet Pythona, jednak bez przykładów implementacji: https://docs.python.org/3/reference/datamodel.html#special-method-names
- Dive Into Python 3 – darmowy podręcznik Pythona online, zawiera m.in. przegląd funkcji specjalnych z omówieniem niektórych spośród nich i niekiedy przykładami implementacji lub użycia: https://diveintopython3.net/special-method-names.html

Inicjalizator

Inicjalizator jest wywoływany podczas tworzenia obiektu; powinien utworzyć wszystkie atrybuty obiektu. Metryka funkcji inicjalizatora ma postać: __init__(self)

```
class Student:

    def __init__(self, /, name, album, *, rok = 1)
        self.name = name
        self.album = album
        self.rok = rok
        self.oceny = {}
        self.ects = 0

student = Student('Nowak', album=39765, rok=3)
```

Konwersja na łańcuch znaków

Są dwa warianty konwersji obiektu na łańcuch znaków:

- __str__ – "oficjalna", wykorzystywana m.in. przez funkcję print() oraz
- __repr__ – robocza, "nieoficjalna", używana w interaktywnym interpreterze

```
class Student:

    def __str__(self)
        return "{}, nr albumu {}".format(
            self.name, self.album)

    def __repr__(self)
        return "<Student, a:{}>".format(self.album)

print(student)  # 'Nowak, nr albumu 39765'
student         # '<Student, a:39765>'
```

Dynamiczne atrybuty

Atrybuty dynamiczne to takie, których wartości powinny być obliczane. Są implementowane przez dwie funkcje, __getattr__ i __setattr__. Python korzysta z tych metod tylko jeżeli obiekt

nie ma atrybutu o danej nazwie, zaś ich metryki mają postać: __getattr__(self, name) oraz__setattr__(self, name, value)

```
class Student:

    def __init__(self, data):
        self.__data = data

    def __getattr__(self, name):
        return self.__data[name]

student = Student({'album':39765, 'lName':'Kowalski'})

student.album   # student.__getattr__('album')
```

Obiekt zachowujący się jak funkcja

Obiekty klasy implementującej funkcję __call__ mogą być wywołane jak funkcje. Jej metryka ma postać: __call__(self, *args, **kwargs)

```
class Student:

    def __call__(self, /, oceny, ects):
        self.__oceny.update(oceny)
        self.__ects += ects

student = Student('Nowak', album=39765)

student({'Matematyka':3}, 4)

# jak student.__call__({'Matematyka':3}, 4)
```

Zadania

Proszę napisać program, który…

1. Definiuje klasę Temperatura, Inicjalizator klasy powinien otrzymać jeden argument (temperaturę) i zapamiętać ją oraz bieżący czas w prywatnych polach. Klasa Temperatura powinna też mieć zdefiniowaną konwersję na typ str (oficjalną i nieoficjalną)

   Wypróbować obiekty klasy Temperatura w interaktywnym interpreterze Pythona (dotyczy to również zadań 2-3)

2. Wykonuje zadania z punktu 1, ale dodatkowo definiuje właściwości tylko do odczytu Celsius i Fahrenheit (można użyć przybliżonej formuły przeliczania, $[F] = 32+2\cdot[C]$ Należy też zdefiniować właściwość tylko do odczytu podającą czas pomiaru
3. * Wykonuje zadania z punktu 2, ale zapisuje temperaturę w prywatnym polu w Kelwinach; Inicjalizator powinien mieć dodatkowo nazwany argument Kelvin – jeżeli Kelvin ma wartość False (powinna to być wartość domyślna), to temperatura jest podana w Celsjuszach i trzeba ją przeliczyć na Kelwiny, a jeżeli True to temperatura jest podana w Kelwinach Należy także odpowiednio zmodyfikować właściwości Celsius i Fahrenheit
4. Definiuje klasę Pomiary;
   Klasa Pomiary powinna zawierać prywatne pole z listą obiektów Temperatura oraz metodę do dodawania pomiaru. Powinna również udostępniać jako właściwość tylko do odczytu kopię listy pomiarów.

   Wypróbować obiekt klasy Pomiary w interaktywnym interpreterze Pythona (dotyczy to również zadań 5-7)

5. * Wykonuje zadania z punktu 4, ale dodatkowo udostępnia w postaci właściwości listy posortowane według wartości oraz czasu pomiaru
6. Wykonuje zadania z punktu 5, ale dodatkowo udostępnia w postaci właściwości minimalną i maksymalną temperaturę
7. * Wykonuje zadania z punktu 6, ale zamiast wartości minimum/maksimum zwraca cały obiekt Temperatura, zawierający wartość minimalną lub maksymalną
8. Wykonuje interaktywnie polecenia dodawania i wyświetlania pomiarów temperatury

   Należy uwzględnić polecenia:
   – dodanie wyniku pomiaru
   – wyświetlenie wszystkich pomiarów
   – * wyświetlenie wyników w Celsjuszach, według czasu pomiaru
   – * wyświetlenie wyników w Fahrenheitach, według czasu pomiaru
   – * wyświetlenie wyników w Celsjuszach, według wartości pomiaru
   – wyświetlenie wartości minimalnej i maksymalnej temperatury
   – * wyświetlenie wartości minimalnej i maksymalnej temperatury i czasu ich wystąpienia
   – koniec trybu interaktywnego