

# Proste a'la Python z definiowaniem własnych klas, atrybutów i metod - Dokumentacja

Krzysztof Pożoga

Styczeń 2022

## 1 Opis funkcjonalności

Implementowanym językiem jest uproszczona wersja Python z definiowaniem własnych klas, atrybutów i metod. Wcięcia zostały zastąpione znakami "{'" i "}"". Wbudowane typy na jakie pozwala język to int oraz string (tylko jako stała tekstowa w wywołaniach funkcji print).

### 1.1 Deklaracje zmiennych

Deklaracja ma formę: *nazwa\_zmiennnej = wartość*.

Aby deklaracja była poprawna po nazwie zmiennej musi pojawić się przypisanie wartości. Zakładamy, że zmienna niebędąca atrybutem klasy żyje do końca bloku, w którym została zadeklarowana.

```
d = 3
```

### 1.2 Klasy

#### 1.2.1 Deklaracje klas

Deklaracja klasy ma formę:

```
class class_name
{
    deklaracje_atrybutow_klasy
    deklaracje_metod
}
```

Atrybuty klasy deklarujemy w taki sam sposób jak zmienne. Są one statyczne dla danej klasy i ich żywotność nie jest powiązana z żadną instancją danej klasy. Dostęp do atrybutu musi być poprzedzony nazwą klasy i znakiem ".".

Dysponując istniejącą instancją klasy możemy deklarować także atrybuty instancji klasy, które od standardowej deklaracji zmiennej odróżniają się tym, że są poprzedzone nazwą instancji klasy i znakiem ".". Do atrybutów statycznych możemy się odwoływać tak samo jak do atrybutów instancji.

### 1.2.2 Deklaracje metod

Deklaracja metody ma forme:

```
def method_name (parameters_list)
{
    kod_wykonywalny
    [return]
}
```

Metody mogą też zwracać pewną wartość za pomocą *return*.

Każda metoda może być wywołana statycznie poprzez *class\_name.method\_name(parameters)* lub dla istniejącej instancji klasy: *class\_instance\_name.method\_name(parameters)*.

Należy wziąć pod uwagę, że w drugim przypadku obiekt będący instancją klasy automatycznie przekazywany jest jako pierwszy parametr wywołania metody.

Dodatkowo metoda o nazwie *\_\_init\_\_(parameters)* nazywana też konstruktorem może być wywołana poprzez *class\_name(parameters)*.

W tym przypadku tworzona jest pusta instancja klasy *class\_name* i konstruktor wywoływany jest jako metoda tejże instancji.

Metody zadeklarowane w przestrzeni globalnej, nazywane też funkcjami, nie są związane z żadną klasą, a co za tym idzie nie mogą być wywołane jako metoda instancji klasy. Przy wywołaniu funkcji nie pojawia się też *class\_name*.

Ponadto przyjmujemy, że funkcja o nazwie *main* jest punktem wejściowym programu.

### 1.2.3 Przykładowa klasa

```
class bike
{
    seial_number = 1
    __init__(self, price)
    {
        self.price = price
        self.serial_number = bike.serial_number
        bike.serial_number +=1
    }
}
```

## 1.3 Petle

### 1.3.1 Petla for

Wykonuje się dla kolejnych liczb z pewnego zakresu.

```
for i = 1 : 10
{
```

```
    kod_wykonywalny  
}
```

### 1.3.2 Petla while

Wykonuje sie dopóki jest spełniany pewien warunek.

*while*  $i < 10$

```
{  
    kod_wykonywalny  
}
```

## 1.4 Instrukcja warunkowa if

Wykona sie jeśli warunek jest spełniony.

*if*  $i > 5$

```
{  
    kod_wykonywalny  
}
```

## 1.5 Operatory

### 1.5.1 Operatory unarne

- minus unarny `"_"`

### 1.5.2 Operatory arytmetyczne

- dodawanie `"+"`
- odejmowanie `"_"`
- mnożenie `"*"`
- dzielenie `"/"`
- dzielenie modulo `"%"`

### 1.5.3 Operatory logiczne

- koniunkcja `"&&"`
- alternatywa `"||"`

#### 1.5.4 Operatory porównania

- równy "=="
- różny "!="
- większy ">"
- większy lub równy ">="
- mniejszy "<"
- mniejszy lub równy "<="

#### 1.5.5 Operatory przypisania

- przypisanie "="
- "+="
- "-="
- "\*="
- "/="
- "%="

### 1.6 Komentarze

Dopuszczalne są komentarze jednolinijkowe zaczynające się od symbolu "#".

### 1.7 Instrukcja wyjścia print

Print jest wbudowana funkcja postaci *print(params)*, gdzie *params* jest lista argumentów będących stringami bądź wartościami "assignable".

Assignable oznacza wszystko, co może być przypisane do zmiennej, a string oznacza ciąg znaków zaczynający i kończący się znakiem ".

Dopuszczalne znaki specjalne:

- koniec linii "\n"
- tabulator "\t"
- backslash "\\"
- cudzysłów "\"

## 2 Gramatyka języka

```
digit = positiveDigit | "0"
positiveDigit = "1" - "9"
letter = "a" - "z" | "A" - "Z" | "_"
id = letter { digit | letter }
comment = "#" { ? any_char ? }
string = " " { ? any_char ? } " "
number = positiveDigit { digit }
minus = "-"
assignOp = "=" | "+=" | "-=" | "*=" | "/=" | "%="
logicOp = "||" | "&&"
relOp = "==" | "!=" | "<" | ">" | "<=" | ">="
addOp = "+" \ | "-"
multOp = "*" | "/" | "%"

program = { classDef | funDef }
classDef = "class" id "{" classBody "}"
classBody = { varDef | funDef }
varDef = fieldId "=" expression
methodDef = "def" id "(" [ Ids ] ")" "{" executable [ret] "}"
Ids = id { "," id }
ret = "return" expression
fieldId = id { "." id }
funCall = fieldId "(" [ arguments ] ")"
arguments = expression { "," expression }
expression = multiplicativeExpr { addOp multiplicativeExpr }
multiplicativeExpr = primaryExpr { multOp primaryExpr }
primaryExpr = [minus] ( number | fieldId | funCall ) | parenthExpr
parenthExpr = "(" expression ")"
assign = fieldId assignOp expression
logicExpr = relExpr { logicOp relExpr }
relExpr = expression relOp expression
if = "if" logicExpr "{" executable "}"
while = "while" logicExpr "{" executable "}"
for = "for" [id "="] expression ":" expression "{" executable "}"
executable = { if | while | for | line }
line = varDef | assign | funCall
```

### 3 Obsługa błędów

Skaner i parser są w stanie wskazać błędny leksykalnie fragment kodu. Błędy w czasie wykonania programu powinny przerwać działanie programu ze zgłoszeniem wyjątku.

### 4 Testy

Testy jednostkowe dla każdego modułu i dla każdego tokenu

### 5 Uruchomienie programu

Projekt będzie pisany w Visual studio 2019 na Win10 x64 jako aplikacja konsolowa. Wykorzystany zostanie język C oraz wbudowany kompilator Visuala.

Konsola przyjmuje input użytkownika lub plik tekstowy do kompilacji /interpretacji.

### 6 Realizacja

Skaner przetwarza wejście na tokeny. Parser analizuje tokeny i na ich podstawie buduje drzewo składniowe.

Interpreter przechodzi po drzewie i na jego podstawie wykonuje kod.

Zaczynając od korzenia przechodzimy rekursywnie wszystkie elementy drzewa.

### 7 Prosty program

```
class samochod
{
    nastepny_numer_produkcyjny = 1337

    def __init__(self, price)
    {
        self.price = price
        self.numer_produkcyjny = samochod.nastepny_numer_produkcyjny
        samochod.nastepny_numer_produkcyjny +=1
    }

    def czyDrozszy(self, sam)
    {
        return self.price > sam.price
    }
}
```

```
}  
  
sam = samochod(1000)  
sam2 = samochod(2000)  
  
if sam.czyDrozszy(sam2) >0  
{  
    sam2 = sam  
}  
  
print(sam2.numer_produkcyjny)
```