AudioScript

Kacper Kamiński

Ogólny opis

Język ten jest językiem skryptowym, opartym na języku Python, którego głównym założeniem jest zautomatyzowanie wielu prostych czynności, jak np. wywoływanie tego samego bloku funkcji na wielu plikach tego samego typu albo szeroko pojętym przetwarzaniu plików dźwiękowych. W tym celu język ten wyposażony jest w statyczne typowanie oraz w dynamiczne importowanie funkcji Pythonowych. Dzięki temu uniknięte zostaną błędy związane z pracowaniem na nieodpowiednich typach obiektów mając jednocześnie możliwość używania obszernych bibliotek Pythona.

Opis języka

Gramatyka EBNF

```
program = declarations, statement-list ;

declarations = empty | dec, lcurly, types-declaration, modules-declaration, rcurly ;

types-declaration = types, empty | identifier, {comma, identifier} ;

modules-declaration = empty | { identifier, lcurly, functions-declarations, rcurly } ;

functions-declarations = empty | { declared-function, semi } ;

declared-function = type-spec, identifier, lparen, declared-args, rparen ;

declared-args = empty | type-spec, {comma, type-spec} ;

type-spec = number | string | identifier ;

var-type = type-spec | var ;

var = "var" ;

types = "Types" ;

number = "number" ;

string = "string" ;

dec = "Declarations" ;

comma = "," ;

statement-list = {statement}* ;

statement = block-statement | assignment-statement, semi | conditional-statement | function-call | empty
```

```
block-statement = lcurly, statement-list, rcurly ;

assignment-statement = variable, assign, (numeric-value | string-value | nill | function-call) ;

conditional-statement = (if-statement | while-statement), block-statement ;

variable-declaration = var-type, identifier, {scomma, identifier} ;

nill = SOME_VALUE ;

empty = ;

variable = indentifier ;

lcurly = "{" ;

rcurly = "}" ;

semi = ";" ;

assign = "=" ;

cond-exp = cond-factor, {higher-logic-operator, cond-factor} ;

cond-value = numeric-value, lower-logic-operator, numeric-value ;

string-value = '"' , { all-characters - '"' }, '"' ;

numeric-value = term, {math-sign, term} ;

term = factor, {binary-math-operator, factor} ;

factor = unary-operator, factor | integer | float | variable | function-call | lparen, numeric-value, r

integer = digit, { digit | zero} ;

float = digit, { digit | zero }, ".", ( digit | zero ), {digit | zero} ;

math-sign = "+" | "-" ;

binary-operator = "*" | "/" | "%" ;

lower-logic-op = "<" | "<=" | ">" | ">=" |
  "==" | "!=" ;

higher-logic-op = "and" | "or" ;

alphabetic-character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;

digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

```
zero = "0" ;

lparen = "(" ;

rparen = ")" ;
```

Część deklaracji typów i funkcji

Jest to pierwsza, opcjonalna część każdego skryptu. Znajdują się w niej deklaracje typów które występować będą w skrypcie, a także deklaracje funkcji które mają zostać dynamicznie dołączone do skryptu. Każda z takich funkcji musi zawierać informację o typach argumentów które przyjmuje, a także o typie zwracanej wartości. Jeżeli jakiś z używanych typów nie jest wbudowany w AudioScript, to musiał być on wymieniony w części deklaracji typów.

Cały blok ma następującą postać:

```
Declarations{
Types: A, B, C;
Modules{
        math{
          A exp(NUMBER);
        }

        builtins{
          print(VAR);
        }
    }
}
```

Idąc od zewnątrz mamy blok `Declarations` który zawiera linię `Types` zawierającą deklarację wszystkich typów "zewnętrznych". Typy te są potencjalnie używane przez załadowane funkcje. W części kodu mogą one zostać użyte do zadeklarowania zmiennych, dla których sprawdzana będzie poprawność ich typu. Przykładowo:

```
A first;
var second;

first = exp(22);
second = first;
```

W powyższym kodzie deklarujemy zmienną typu `A`. W drugiej linii przypisujemy do niej wynik wykonania `exp(22)`. Interpreter wie z deklaracji funkcji w bloku `Declarations` że zwraca ona wartość typu `A`, dlatego operacja zostaje wykonana. W trzeciej linii widzimy jednakże próbę przypisania do `second`, wartości przechowywanej w `first`. Interpreter zgłosi tutaj błąd, ponieważ nie może zostać przypisana do zmiennej typu `VAR`.

```
TypeError: TypeError: Expected VAR and got A
```

Idąc dalej mamy blok `Functions` w którym zawierają się bloki modułów. Moduł identyfikowany jest poprzez nazwę i ograniczony jest nawiasami klamrowymi. W samym bloku znajdują się deklaracje funkcji mające postać `"zwracana wartość" "nazwa funkcji"(typy argumentów);` Każda z funkcji zostanie dołączona dynamicznie do skryptu, na podstawie nazwy modułu Pythonowego w który się znajduje. Przykładowo do skryptu w którym znajduje się poniższa deklaracja modułu zaimportowana zostanie funkcja funkcja `floor` z modułu `math`.

```
math{
```

```
number floor(number);
}
```

Dołączanie funkcji przez generator możliwe jest dzięki funkcji `__import__()` w Pythonie która umożliwia dynamiczne załadowanie modułu. Następnie używając funkcji `getattr` możemy dostać pożądaną funkcję. Dla naszego wcześniejszego przykładu sprowadzi się to do poniższego kodu.

```
math = __import__("math")
floor = getattr(math, "floor")
```

Nazwy funkcji(nawet w różnych modułach) muszą być unikalne w celu uniknięcia konfliktów.

Sprawdzanie typu zmiennych zostanie zaimplementowane z użyciem funkcji `type()`. Zwraca ona typ podanego obiektu.

Część kodu

W drugiej, również opcjonalnej części skryptu znajduje się kod do wykonania. Część ta zaczyna się od pierwszej linii po bloku `Declarations`. Może składać się z definicji funkcji, operacji, deklaracji zmiennych, wywołań funkcji itp.

Definicja funkcji

W bloku kodu możliwe jest zdefiniowanie nowych funkcji z użyciem słowa kluczowego `function`. W przeciwieństwie do dynamicznie dołączanych funkcji z bloku `Declarations` nie muszą one określać wprost typu przyjmowanych argumentów oraz zwracanych wartości. Blok funkcji składa się ze słowa kluczowego `function` po którym następuje jej nazwa, lista argumentów w nawiasie oraz nawiasy klamrowe w których mogą znajdować się dalsze instrukcje. Nazwy funkcji musza być unikalne. `function foo(a, b, c){...}`

Wbudowane typy

Sam język zawierać będzie następujące wbudowane typy:

- NUMBER - Typ ten reprezentuje liczby zmiennoprzecinkowe.
- STRING - Typ ten reprezentuje ciąg znakowy.

Instrukcje

Instrukcje składają się ze zmiennych, operatorów lub/i wywołań funkcji. Każda instrukcja musi być zakończona średnikiem.

Matematyczne operatory binarne

- `+` : dodawanie
- `-` : odejmowanie
- `/` : dzielenie
- `*` : mnożenie

Operatory te mogą być używane na zmiennych typu `NUMBER`(operatory `+` oraz `*` mogą być także użyte na zmiennych typu STRING.

Poniżej widać kod prezentujący wykorzystanie wszystkich operatorów:

```
var a, b, c, d, e, f;
a = 22.222;
b = 3;
c = a + b;
d = a - b;
e = a * b;
f = a / b;
```

A także wynik działania powyższego kodu:

```
SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
 NUMBER: NUMBER
 STRING: STRING
    VAR: VAR
      a: <VarSymbol(name='a', type='VAR')> = 22.222
      b: <VarSymbol(name='b', type='VAR')> = 3
      c: <VarSymbol(name='c', type='VAR')> = 25.222
      d: <VarSymbol(name='d', type='VAR')> = 19.222
      e: <VarSymbol(name='e', type='VAR')> = 66.666
      f: <VarSymbol(name='f', type='VAR')> = 7.407333333333334
```

Operatory logiczne

- `==` : operator równości
- `!=` : operator nierówności
- `<` : operator mniejszości
- `>` : operator większości
- `<=` : operator mniejszy lub równy
- `>=` : operator większy lub równy
- `and` : koniunkcja logiczna
- `or` : alternatywa logiczna

Operatory te mogą być używane na zmiennych typu `number`. Operatory `!=` oraz `==` mogą być także używane na zmiennych typu `string`, porównywane jest wtedy czy ciągi znakowe są lub nie są identyczne. Mogą być one użyte także ze zmienną typu `null`. Wynikiem porównania `null` z jakimkolwiek innym obiektem poza `null` jest zawsze fałsz. Operatory logiczne mogą być używane w kontekście instrukcji warunkowej `if` oraz instrukcji pętli `while`, przykładowo dla poniższego kodu:

```
Declarations{
Modules{
        builtins{
            print(VAR);
        }
    }
```

```
}
var a, b;
a = 0;
b = -11;

if(a == 0 and b == -11)
    print("if and and work");

while(a == 22 or b < 0){
    print("while and or also work");
    b = b + 1;
}
```

Wynik działania możemy zaobserwować na poniższym zdjęciu:

```
 if and and work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work
 while and or also work


 SCOPE (SCOPED SYMBOL TABLE)
 ===========================
 Scope name     : global
 Scope level    : 1
 Enclosing scope: None
 Scope (Scoped symbol table) contents
 -----------------------------------
  NUMBER: NUMBER
  STRING: STRING
     VAR: VAR
   print: <ExternalFunctionSymbol(name=print, arguments types=[VAR], return type=NULL)>
       a: <VarSymbol(name='a', type='VAR')> = 0
       b: <VarSymbol(name='b', type='VAR')> = 0
```

Instrukcja warunkowa `if` wykona kod zawarty w swoim bloku instrukcji jeżeli wyrażenie w jej nawiasach zostanie ewaluowane jako prawda. W przeciwnym razie blok `if` jest pomijany.

Instrukcja pętli `while` będzie wykonywała kod zawarty w swoim bloku instrukcji, dopóki wyrażenie w jej nawiasach zostanie ewaluowane jako prawda.

Inne

`=` : operator przypisania
`#` : komentarz

Operator przypisania służy do przypisania wartości, bądź obiektu do zmiennej. Symbol `#` mówi że wszystko co występuje po nim w danej linii ma zostać pominięte.

Deklaracja zmiennych

Deklaracja odbywa się albo poprzez podanie jednego z zadeklarowanych w części `Declarations` typów, a następnie nazwy zmiennej, lub nazwy wielu zmiennych tego typu oddzielonych od siebie przecinkami oraz zakończenie deklaracji średnikiem. Przykładowo `A zmienna; Lista x, y, z;`. Deklaracja zmiennej może się odbyć także poprzez podanie słowa kluczowego `var`, a następnie podanie nazwy, bądź wielu nazw i średnika. Przykładowo `var a, b, c;`.

W pierwszym przypadku przy przypisywaniu do danej zmiennej wartości, bądź obiektu sprawdzony zostanie typ przypisywanej wartości lub obiektu i w razie niezgodności z typem zmiennej zgłoszenie błędu. Także przy przekazywaniu zmiennej jako argument funkcji zgłoszony zostanie błąd jeżeli nie zgadza się ona z typem argumentu.

W drugim przypadku otrzymamy zmienną która może przechowywać jeden z typów podstawowych języka. Jest to rozwiązanie analogiczne co w języku Python. Każda taka zmienna może przechowywać dowolny obiekt jeżeli ma on jeden z podstawowych typów, przykładowo:

```
var x;
x = 3; # Przypisujemy wartość 3 (typ number)
x = "Abc" # A następnie przypisujemy ciąg znakowy "Abc"
```

Do takiej zmiennej nie da się jednakże przypisać jednego z typów obcych z bloku `Declarations`.

```
B a;
var x;
a = function2();
x = a; # ERROR
```

Za jednym zamachem możemy także zadeklarować wiele zmiennych danego typu:

```
var x, y, b, s;
A po, pp, pq, pw;
```

Niedozwolone jest jednakże inicjalizowanie zmiennej, należy najpierw dokonać deklaracji, a następnie przypisać wartość:

```
var x= 4; # BŁĄD
var y;
y = 5; # OK
```

Przykładowo dla:

```
var x, y, z, t, p;
x = 1;
y = 2;
```

Otrzymujemy następującą tablicę symboli:

```
SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
 NUMBER: NUMBER
 STRING: STRING
    VAR: VAR
      x: <VarSymbol(name='x', type='VAR')> = 1
      y: <VarSymbol(name='y', type='VAR')> = 2
      z: <VarSymbol(name='z', type='VAR')> = None
      t: <VarSymbol(name='t', type='VAR')> = None
      p: <VarSymbol(name='p', type='VAR')> = None
```

Wywołanie funkcji

Wywołanie funkcji odbywa się poprzez podanie nazwy funkcji, po czym podaniu argumentów funkcji.

```
var x = 2;
foo(x, 2, "aa");
```

Wartość zwracana przez funkcję może być przypisana do zmiennej.

```
var z = foo(2);
```

Argumentami funkcji mogą być dowolne wyrażenia których typ wartości po ewaluowaniu zgadza się z zadeklarowanym typem argumentu.

```
Declarations{
    Modules{
        math{
            VAR exp(VAR);
        }
    }
}
def foo(){
return -15;
}
var x;
x = exp(15 + foo());
```

Wynik:

```
SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
 NUMBER: NUMBER
 STRING: STRING
    VAR: VAR
    exp: <ExternalFunctionSymbol(name=exp, arguments types=[VAR], return type=VAR)>
    foo: <FunctionSymbol(name=foo, arguments=[])>
      x: <VarSymbol(name='x', type='VAR')> = 1.0
```

Zakres nazw

Każdy blok nawiasów klamrowych tworzy nowy zakres widoczności nazwy, zawierający się w tej w której nawiasy zostały umieszczone. Najbardziej zewnętrzny zakres jest przestrzeń globalna.
Nazwy(np. zmiennych lub funkcji) szukane są zaczynając od obecnego zakresu i w przypadku nie znalezienia ich, następuje próba znalezienia ich w szerszym zasięgu(aż do przestrzeni globalnej).
Nazwy w zakresach na niższym poziomie hierarchii nie są dostępne w bardziej zewnętrznych przestrzeniach nazw. Przykładowo:

```
var a;
{
var x = 1;
}
a = x; # Błąd, nie znaleziono x!
```

Nazwy zdefiniowane w bardziej zewnętrznych przestrzeniach nazw są jednakże dostępne w wewnętrznych przestrzeniach nazw.

```
var a = 3;
{
var x = a; # Dozwolone, a jest dostępne w tym zasięgu
}
```

Nazwy(ale nie obiekty) przestają być dostępne po wyjściu z zakresu w którym zostały zadeklarowane.

Występuje także przysłanianie nazw zewnętrznych przez nazwy wewnętrzne. Przykładowo dla kodu:

```
var x;
x = 5;
{
  var x;
  x = 2;
}
var z;
z = 1;
```

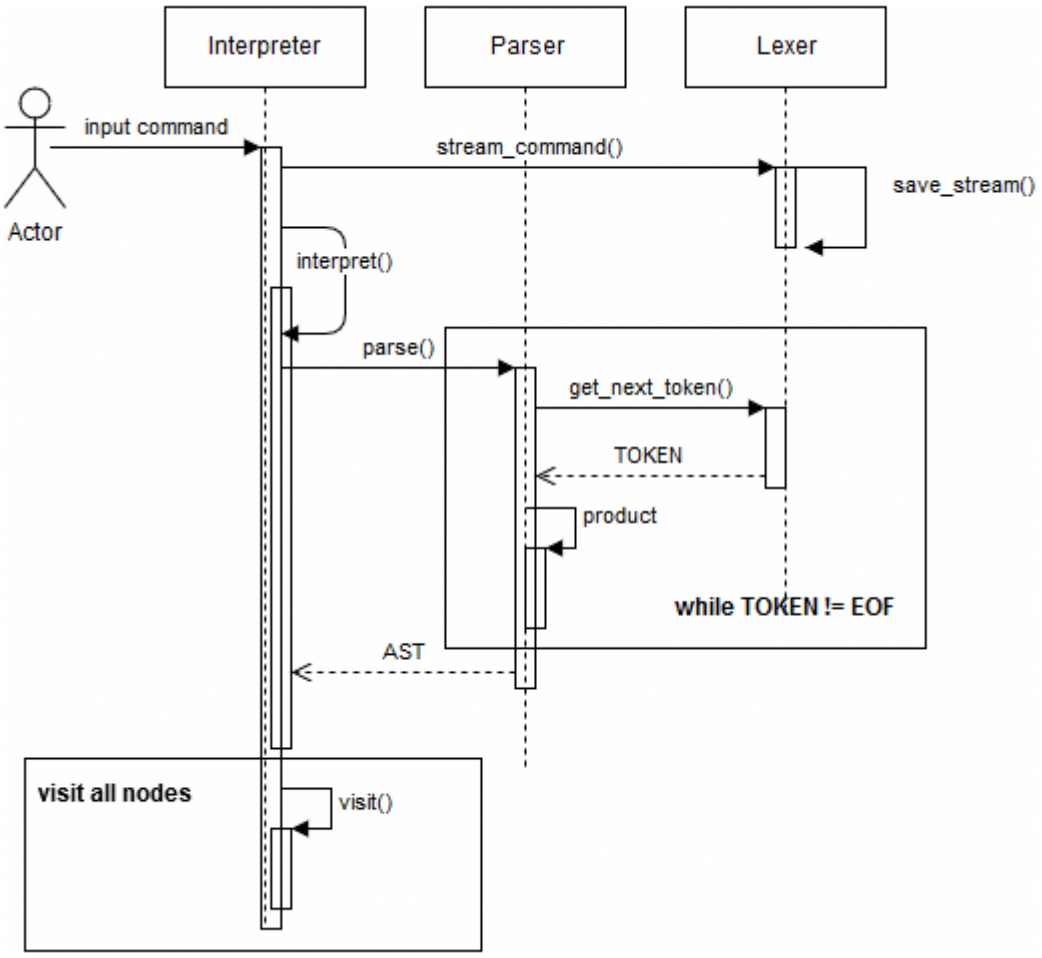
Otrzymujemy podane tablice symboli:

```
SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
-----------------------------------
 NUMBER: NUMBER
 STRING: STRING
    VAR: VAR
      x: <VarSymbol(name='x', type='VAR')> = 5
      z: <VarSymbol(name='z', type='VAR')> = 1
```

Architektura Interpretera

Składać się on będzie z 3 modułów: leksera mającego na celu podzielenie czytanego tekstu (przekazanych poleceń) na tokeny; parsera przyjmującego tokeny tworzone przez lekser i tworzącego na ich podstawie drzewo składniowe oraz interpretera który przyjmować będzie tworzone drzewo i wykonywał na jego podstawie odpowiednie akcje semantyczne.

Na poniższym diagramie zaprezentowana jest komunikacja odpowiednich modułów.



Lekser

Moduł ten odpowiada za dzielenie czytanego strumienia tekstu na tokeny. Każdy token jest jednym z symboli terminalnych w gramatyce naszego basha. Lekser czyta znak po znaku strumień tekstu który zostanie do niego przekazany i kiedy uda mu się dopasować uzyskany ciąg do jednego z wyrażeń regularnych definiujących któryś z Tokenów zostaje on utworzony i zwrócony.

Parser

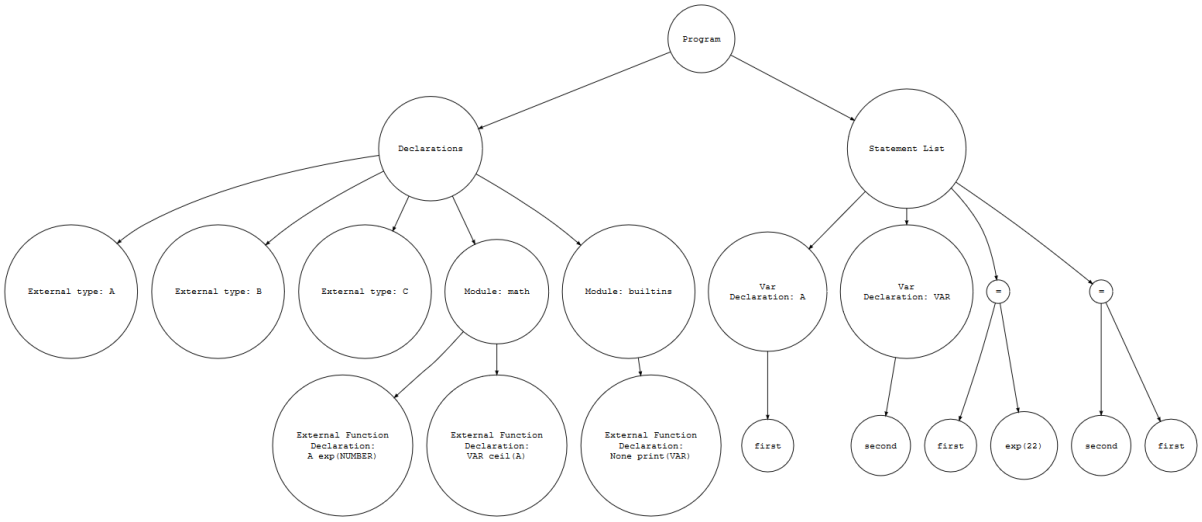
Moduł ten odpowiada za budowę AST - abstrakcyjnego drzewa składniowego. Będzie on typu rekursywnie-zstępującego. Parser przyjmuje po kolei tokeny zwracane przez lekser a następnie na podstawie gramatyki naszego basha buduje kolejne węzły drzewa. Każdy węzeł drzewa odpowiednia jakiejś części składni języka. Przykładowo dla poniższego kodu:

```
Declarations{
Types: A, B, C;
Modules{
        math{
        A exp(NUMBER);
        VAR ceil(A);
```

```
        }
        builtins{
        print(VAR);
        }
    }
}
A first;
var second;

first = exp(22);
second = first;
```

Wygenerowane zostanie poniższe drzewo:



Interpreter

Moduł ten odpowiada za przejście po przekazanym abstrakcyjnym drzewie składniowym i wykonanie odpowiednich akcji semantycznych.

Interpreter przechodzi po drzewie AST stosując metodę przechodzenia w głąb, dlatego też jego działanie zaczyna się od korzenia drzewa AST.

“‘ def interpret(): tree = parser.parse() // Interpreter pobiera z parsera drzewo AST visit(tree) // Następnie zaczyna przechodzenie po nim ”’

Każdemu typowi węzła w naszym drzewie odpowiadać będzie jedna funkcja wykonująca odpowiednią akcję semantyczną.

Przykładowo dla wcześniej wymienionego węzła Assign:

”‘def visit_Assign(): var_name = node.left.value value = node.right.value value_type = node.right.type variables_scope.assign(var_name, value, value_type)”’

Jak widać powyżej z lewego węzła czytana jest nazwa zmiennej, z prawego typ wartości przypisanej do zmiennej oraz jej wartość. Następnie wartości te zostają zapisane w jakiejś strukturze przechowującej zmienne w naszej obecnej przestrzeni zmiennych.

Wyjątki

Podczas parsowania lub wykonywania skryptu może dojść do sytuacji błędnych, nieprzewidzianych przez programistę. Sytuacje takie zgłaszane będą przez błąd. Wyjątki zostają wyświetlone w specjalnym komunikacie, po czym zatrzymywane jest wykonanie skryptu. Poniżej znajduje się lista możliwych wyjątków.

- `SyntaxError` - Zgłaszany jest on przez parser w momencie kiedy natrafia on na fragment kodu, który nie jest poprawny gramatycznie.
- `ArithmeticError` - Zgłaszany jest on kiedy dojdzie do niepoprawnej operacji matematycznej, takiej jak dzielenie przez zero.
- `ImportError` - Zgłaszany jest on kiedy nie uda się znaleźć i dołączyć jednej z funkcji w bloku `Declarations`
- `NameError` - Zgłaszany jest on kiedy podana nazwa nie występuje w danej przestrzeni nazw.
- `TypeError` - Zgłaszany jest on kiedy przypisujemy do zmiennej wartość lub obiekt która nie jest zgodna z typem danej zmiennej, bądź kiedy podajemy zmienną o niepoprawnym typie jako argument funkcji.
- `PythonError` - Zgłaszany jest on kiedy jedna z dołączonych funkcji zgłosi jakiś wyjątek.

Uruchamianie skryptu

Aby uruchomić skrypt, należy uruchomić interpreter i przekazać ścieżkę do niego w argumencie, przykładowo: `python Interpreter code`.