5. Learning More - Documentation and Errors

5.1. Documentation

Now that you've got a basic grasp of Pyth, the best way to learn more about the language is via the documentation. Pyth's documentation is located on Github, and is updated continually as the language evolves.

While the documentation is a good first step towards understanding how a function works, the only way to get a full understanding is by using it. Write programs using that function, and it'll become clear.

5.2. Errors

There are two levels of errors you might get: errors at the Pyth level, and errors at the Python level.

5.2.1 Pyth Errors

Currently, there are only 3 types of errors implemented in Pyth: token not implemented errors, unsafe input errors, and wrong type errors. I'll go through these one by one.

Token not implemented errors occur when you try to use an unimplemented token. They look like this:

```
=================== 6 chars =====================
5.@1 1
=================================================
Traceback (most recent call last):
  File "pyth.py", line 454, in <module>
    py_code_line = general_parse(code)
  File "pyth.py", line 38, in general_parse
    parsed, code = parse(code)
  File "pyth.py", line 105, in parse
    raise PythParseError(active_char, rest_code)
extra_parse.PythParseError: .@ is not implemented, 4 from the end.
```

These are most commonly caused by using non-ASCII characters in Pyth code outside of strings, and by ending floating point numeric literals with a `.`, which is confused with tokens of the form `._`, as seen above.

Unsafe input errors occur when you try to use `$` (the python code injection character) in the online compiler / executor or when the `-s` flag (safe mode) is enabled. Using `$` is a security hole, and is therefore disabled online.

Wrong type errors are the most common type of error. Most functions are defined via type overloading, where the function does something entirely different depending on what types are given as input. However, not all combinations of types have an associated functionality. For instance:

```
=================== 9 chars =====================
@"abc"1.5
=================================================
Pprint("\n",lookup("abc",1.5))
=================================================
Traceback (most recent call last):
  File "pyth.py", line 478, in <module>
  File "<string>", line 4, in <module>
  File "/app/macros.py", line 84, in lookup
macros.BadTypeCombinationError:
Error occured in function: @
Arg 1: 'abc', type str.
Arg 2: 1.5, type float.
```

The relevant part to look at is the last four lines. The fourth to last line indicates that an error as caused due to a bad type combination. The third to last line indicates that the error occurred in the `@` function, and the rest of the lines indicate that the error occurred because `@` was called with a string as its first argument and a float as its second argument, for which it has no defined functionality.

5.2.2 Python Errors

Occasionally you will get an error such as:

```
=================== 4 chars =====================
@""1
=================================================
Pprint("\n",lookup("",1))
=================================================
Traceback (most recent call last):
  File "pyth.py", line 478, in <module>
  File "<string>", line 4, in <module>
  File "/app/macros.py", line 73, in lookup
ZeroDivisionError: integer division or modulo by zero
```

that doesn't fall into any of the previous categories. At this point, the best solution is to simply try different variants on the program in an attempt to understand how it ticks. If that doesn't work, the only remaining recourse is to look at the code itself.

As indicated by the traceback, looking at line 73 of `macro.py` we see that `@` attempts to perform a lookup in the string at the index given by the second argument modulus the length of the string. Since the length of the string is zero, Python's modulus operator will fail and throw a `ZeroDivisionError`.