

How Python Works

15-110 – Friday 01/17

Learning Objectives

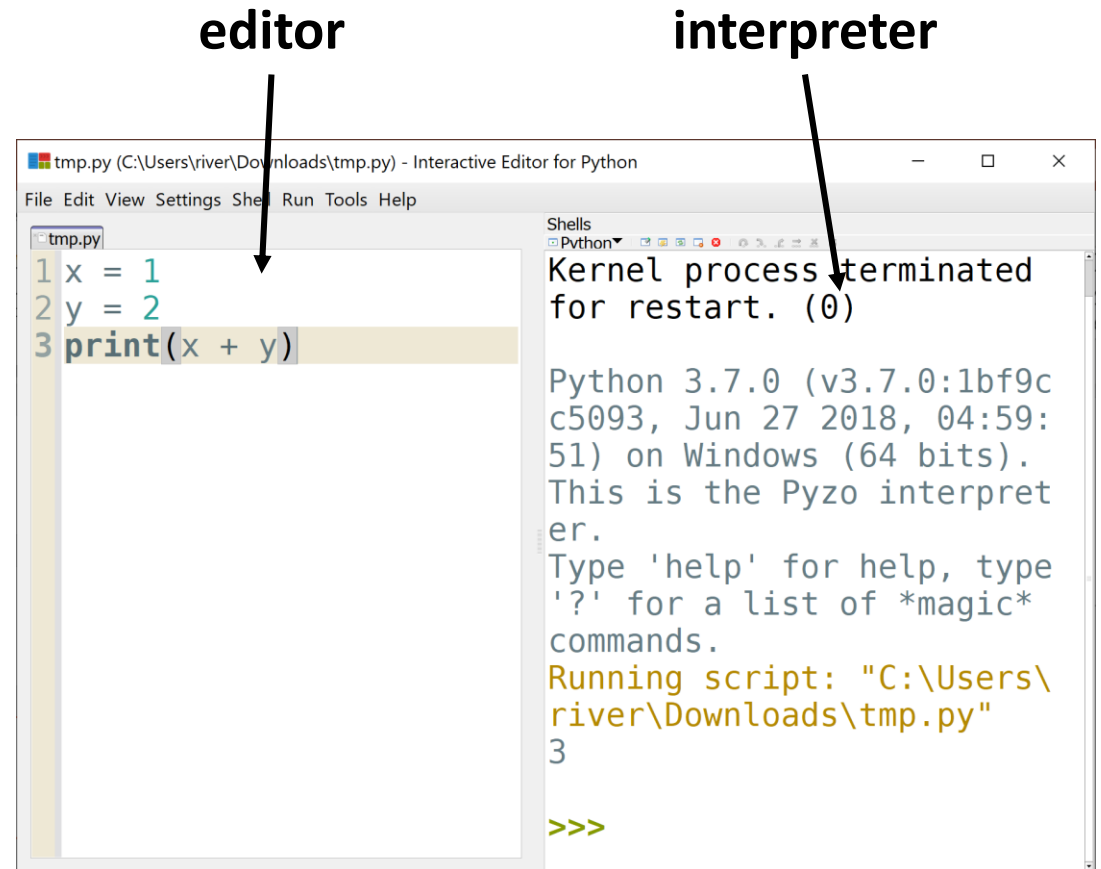
- Recognize the steps of the process that converts python code into instructions a computer can execute
- Understand how and when different types of errors occur in python code

The Pyzo IDE Has an Editor and an Interpreter

We mostly use two parts of the Pyzo IDE while writing code- the **editor** and the **interpreter**.

The **editor** is just a normal text editor. When we save text, it is saved to a .py file, but this is still just normal text.

The **interpreter** does the actual work of converting our Python text into instructions the computer can run. This happens when you click 'Run File as Script'.



Sidebar: How Files Work

Your computer uses file and folders to organize data content locally (on the hardware).

A **file** is a single piece of content – a document, or a picture, or a song, or Python code.

A **folder** is a structure that can hold 0+ files, as well as other folders. Folders can be nested for further organization. Folders let you manage files directly.

You'll create many files (mostly .pdf and .py files) for this class. We recommend that you make a 15-110 folder to hold all your work.

The Interpreter Turns Python Code to Bytecode

Python code is **abstracted** – it's written at a level humans can understand. But this is too high-level for a computer to follow instructions directly.

A computer does know how to follow a small set of instructions that are built into its hardware. These instructions are called **bytecode**.

The job of the **interpreter** is to translate your python code into bytecode, which the computer can then run.

To do this translation, the interpreter **tokenizes**, **parses**, and **translates** the code.

Tokenizing Splits Text into Tokens

First, the interpreter takes a big set of text (the Python program) and breaks it into **tokens**.

```
x = 5  
number = x-2
```

It identifies natural break points based on the grammar of the language. For example, in the code to the right, the tokens produced would be:

x, =, 5, newline, number, =, x, -, 2

Parsing Groups Tokens by Task

Next, the interpreter **parses** the sequence of tokens into a structured format called a parse tree.

x, =, 5, newline,
number, =, x, -, 2

This tree groups together tokens that are part of the same action.

For example, given the tokens to the right, the interpreter would recognize that **=** is an action taken with **x** as the target variable and **5** as the value.

Tokenizing and Parsing Errors are Syntax Errors

The first two steps – tokenizing and parsing – are based on the Python language's **syntax**. Syntax is a set of rules for how code instructions should be written.

If the interpreter runs into an error while tokenizing or parsing, it calls that a **syntax error**. You get a syntax error when the code you provide does not follow the rules of the Python language's syntax.

Examples of Syntax Errors

Most syntax errors are called **SyntaxErrors**, which make them easy to spot. For example:

```
x = @      # @ is not a valid token
4 + 5 = x  # the parser stops because it doesn't follow the
rules
```

There are two special types of syntax errors: **IndentationError** and incomplete error.

```
    x = 4    # IndentationError: whitespace has meaning
print(4 + 5 # Incomplete Error: always close parentheses
```

Bytecode is a Simple Language

Once code has been parsed, the interpreter can translate it into a language, **bytecode**. Bytecode is composed of a small list of instructions the computer knows how to perform. You can find a full list here:

docs.python.org/3/library/dis.html#python-bytecode-instructions

Bytecode instructions are very simple and structured. Each line has a single instruction – a command name, and (sometimes) a number.

Because the language is so simple, it relies on additional components to run: **tables** of values, which form the program's memory, and a **stack**, which keeps track of the program's state as it runs.

Example: Value Tables and the Stack

For example, consider the following program:

```
x = 5
y = 7
z = x + y
```

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	
1	y	
2	z	

The computer stores of all the values used by the program in two tables: a **Literal Value** table and a **Variable** table.

It also uses a **Stack**, where it stores information it needs to execute commands. The stack is like your working memory.



Example: Bytecode Instructions

The instructions the computer will use for our example are as follows.

`LOAD_CONST` and `LOAD_NAME` are used to move information from a table onto the stack.

`STORE_NAME` is used to move information from the stack into the variable table.

`BINARY_ADD` will add together the top two values on the stack and replace them with the result. `BINARY_SUBTRACT` does the same, but with subtraction.

Example: Bytecode Execution

Put all of this together, and the program below is translated to the bytecode on the right.

```
x = 5
y = 7
z = x + y
```

Let's walk through what the bytecode does.

```
LOAD_CONST      0
STORE_NAME
LOAD_CONST      1
STORE_NAME
LOAD_NAME       0
LOAD_NAME       1
BINARY_ADD
STORE_NAME      2
```

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	
1	y	
2	z	

Stack

Bytecode-Running Errors are Runtime Errors

If an error occurs as bytecode is being executed, it's called a **runtime error**. That's because the error occurs as the code is running!

Runtime errors have many different names in Python. Each name says something about what kind of error occurred, so reading the name and text can give you additional information about what went wrong.

Examples of Runtime Errors

```
print>Hello) # NameError: used a missing variable
```

```
print("2" + 3) # TypeError: illegal operation on types
```

```
x = 5 / 0 # ZeroDivisionError: can't divide by zero
```

We'll see more types of runtime errors as we learn more Python syntax.

Other Errors are Logical Errors

If we manage to translate Python code into bytecode and it runs completely, does that mean it's correct?

Not necessarily! **Logical errors** can occur if code runs but produces a result that was not what the user intended. The computer can't catch logical errors, because the computer doesn't know what we intend to do.

Logical errors will be the hardest to find and fix. We'll talk more about addressing them later.

Activity: Step Through Bytecode

Task: we've translated a simple program into bytecode and set up its initial tables.

Walk through the bytecode to determine what values are held in variables **a**, **b**, and **c** at the end of the code.

Submit your answer on Piazza when you're done.

Note: subtract the higher element on the stack from the lower element.

```
LOAD_CONST 0  
STORE_NAME 0
```

```
LOAD_NAME 0  
LOAD_CONST 1  
BINARY_SUBTRACT  
STORE_NAME 1
```

```
LOAD_NAME 0  
LOAD_NAME 1  
BINARY_ADD  
STORE_NAME 2
```

Literal Table	
id	value
0	6
1	2

Variable Table		
id	name	value
0	a	
1	b	
2	c	

Stack	

Learning Objectives

- Recognize the steps of the process that converts python code into instructions a computer can execute
- Understand how and when different types of errors occur in python code