

How Python Works

15-110 – Friday 09/11

Learning Objectives

- Recognize that the process of **tokenizing, parsing, and translating** converts Python code into instructions a computer can execute
- Interpret and trace basic **bytecode** instructions
- Recognize how the different types of **errors** are raised at different points in the Python translation process

Tokenizing, Parsing, Translating

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 directly.

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

Bytecode is machine language for an imaginary, simplified computer.

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

To do this, the interpreter **tokenizes**, **parses**, and then **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 = 15
coord = 3*(x-2.7)
```

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, =, 15, newline, coord, =, 3, *, (, x,  
-, 2.7, )
```

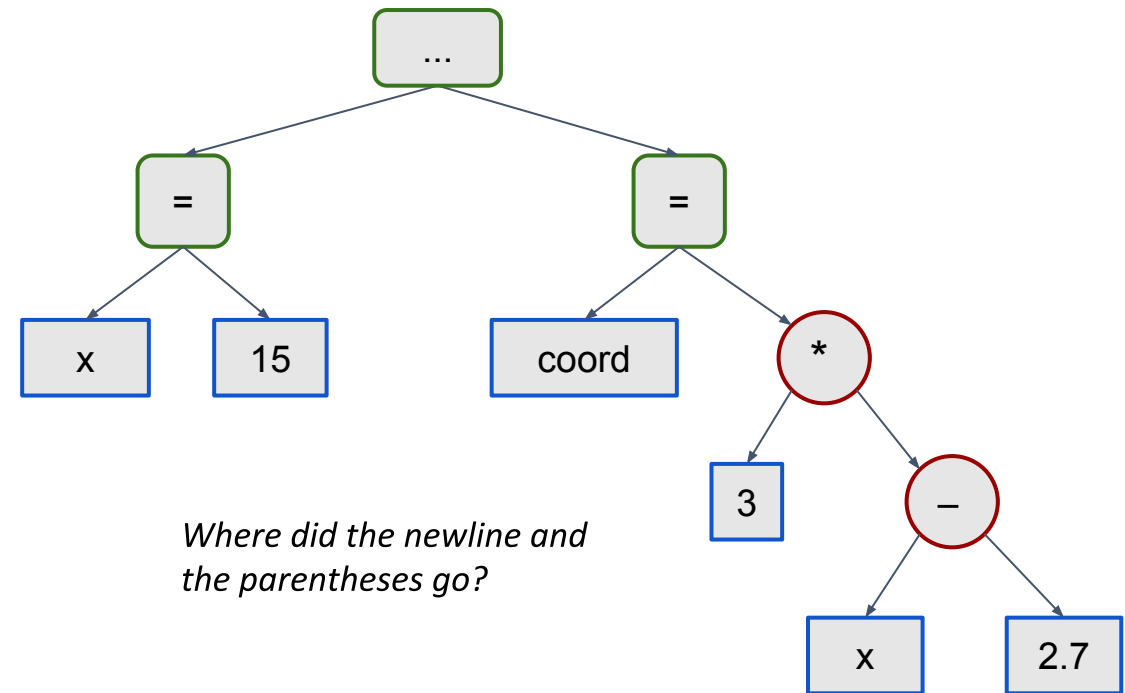
Parsing Groups the Tokens by Task

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

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 **15** as the value.

x, =, 15, newline,
coord, =, 3, *, (, x, -, 2.7,)



Where did the newline and the parentheses go?

Translating Parse Trees to Bytecode

Once code has been parsed, the interpreter can translate it into a different language, **bytecode**.

Bytecode is composed of a small list of instructions that can be run by **any** computer. This means that a program you write on your laptop will run the same way on a school computer. Bytecode translates directly to **machine code**, which is built in to every computer's hardware.

You can find a full list here of the bytecode instructions associated with Python here:

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

Bytecode

Bytecode is a Simple Language

Bytecode instructions are very simple and structured. Each line has a single instruction, which consists of a command name and (sometimes) a number. For example:

```
LOAD_CONST 0 # load the literal (constant) at loc. 0
```

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

Literal and Variable Tables, and Value Stack

For example, consider the following program:

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

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

It also uses a **Value Stack**, where it stores information temporarily for use with instructions. The stack is like your working memory.

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	
1	y	
2	z	



Example: Bytecode Instructions

Here are the instructions the computer will use for our example:

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

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

`BINARY_ADD` will add together the top two values on the value 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 # 5
STORE_NAME 0 # x

LOAD_CONST 1 # 7
STORE_NAME 1 # y

LOAD_NAME 0 # x
LOAD_NAME 1 # y
BINARY_ADD
STORE_NAME 2 # z
```

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	
1	y	
2	z	

Value Stack

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 # 5`
`STORE_NAME 0 # x`

`LOAD_CONST 1 # 7`
`STORE_NAME 1 # y`

`LOAD_NAME 0 # x`
`LOAD_NAME 1 # y`
`BINARY_ADD`
`STORE_NAME 2 # z`

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	
1	y	
2	z	

5
Value Stack

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 # 5
   STORE_NAME 0 # x

   LOAD_CONST 1 # 7
   STORE_NAME 1 # y

   LOAD_NAME 0 # x
   LOAD_NAME 1 # y
   BINARY_ADD
   STORE_NAME 2 # z
```

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	
2	z	

Value Stack

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 # 5
STORE_NAME 0 # x
→ LOAD_CONST 1 # 7
STORE_NAME 1 # y
LOAD_NAME 0 # x
LOAD_NAME 1 # y
BINARY_ADD
STORE_NAME 2 # z
```

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	
2	z	

7
Value Stack

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 # 5
STORE_NAME 0 # x

LOAD_CONST 1 # 7
→ STORE_NAME 1 # y

LOAD_NAME 0 # x
LOAD_NAME 1 # y
BINARY_ADD
STORE_NAME 2 # z
```

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	7
2	z	

Value Stack

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 # 5`
`STORE_NAME 0 # x`

`LOAD_CONST 1 # 7`
`STORE_NAME 1 # y`

→ `LOAD_NAME 0 # x`
`LOAD_NAME 1 # y`
`BINARY_ADD`
`STORE_NAME 2 # z`

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	7
2	z	

5
Value Stack

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 # 5`
`STORE_NAME 0 # x`

`LOAD_CONST 1 # 7`
`STORE_NAME 1 # y`

`LOAD_NAME 0 # x`
`LOAD_NAME 1 # y`
`BINARY_ADD`
`STORE_NAME 2 # z`

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	7
2	z	

7
5
Value Stack

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 # 5`
`STORE_NAME 0 # x`

`LOAD_CONST 1 # 7`
`STORE_NAME 1 # y`

`LOAD_NAME 0 # x`
`LOAD_NAME 1 # y`
→ `BINARY_ADD`
`STORE_NAME 2 # z`

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	7
2	z	

12
Value Stack

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 # 5`
`STORE_NAME 0 # x`

`LOAD_CONST 1 # 7`
`STORE_NAME 1 # y`

`LOAD_NAME 0 # x`
`LOAD_NAME 1 # y`

`BINARY_ADD`
`→ STORE_NAME 2 # z`

Literal Table	
id	value
0	5
1	7

Variable Table		
id	name	value
0	x	5
1	y	7
2	z	12

Value Stack

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.

Work with your breakout group to trace through the code.

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	

Value Stack

Functions and the Call Stack

Recall our discussion about variable scope. Scope exists because of the **Call Stack**. In fact, the Call Stack is what makes it possible for us to nest function calls!

Consider the code to the right. When the function `inner` is called inside of `outer`, the current **state** of `outer` is put on the call stack.

When `inner` returns a value, that value is sent back to the level of the stack with the `outer` function's state, to resume running that function.

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
print(outer(4))
```

Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

outer(4)

Global Variable Table

[] ; outer(4)

Call Stack

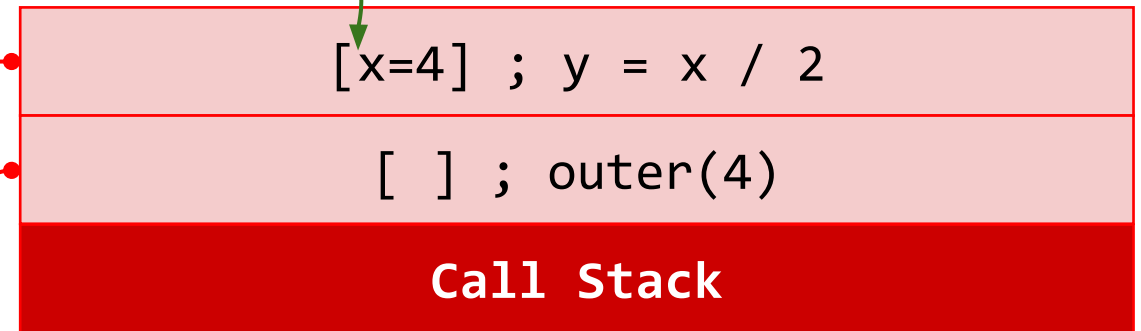
Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
outer(4)
```

Local Variable Table



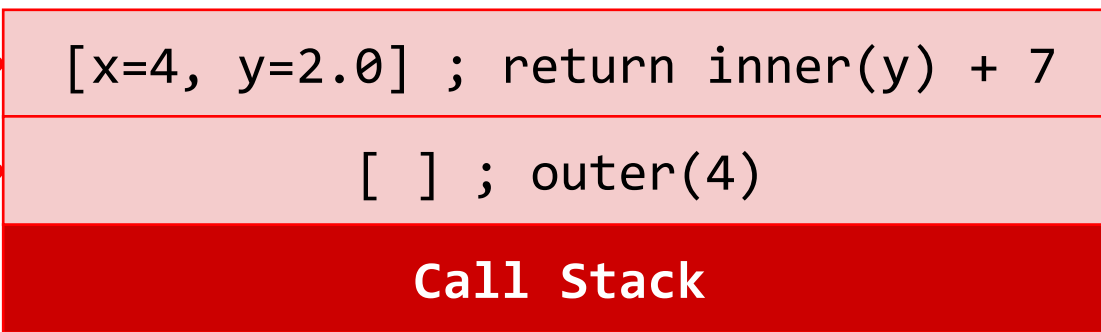
Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
outer(4)
```

Local Variable Table



[x=4, y=2.0] ; return inner(y) + 7

[] ; outer(4)

Call Stack

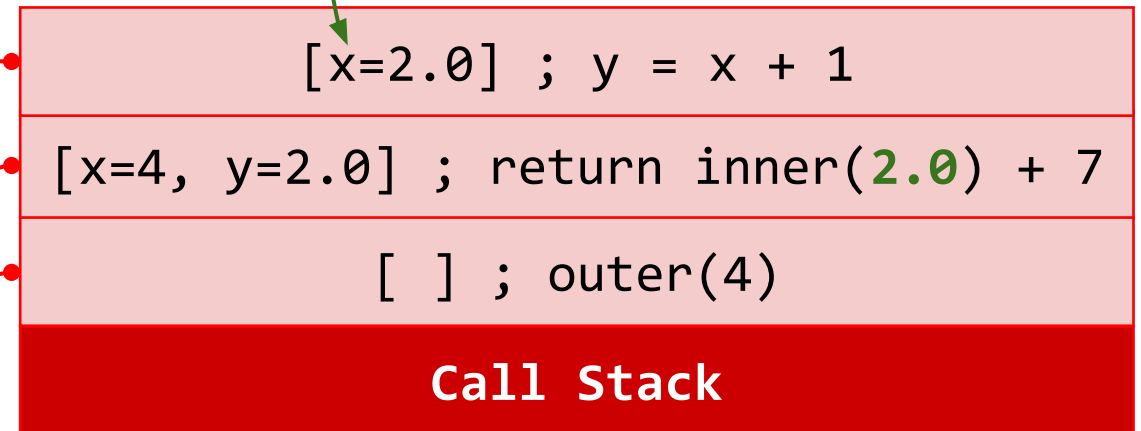
Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
outer(4)
```

Local Variable Table



Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
outer(4)
```

Local Variable Table

[x=2.0, y=3.0] ; return y

[x=4, y=2.0] ; return inner(2.0) + 7

[] ; outer(4)

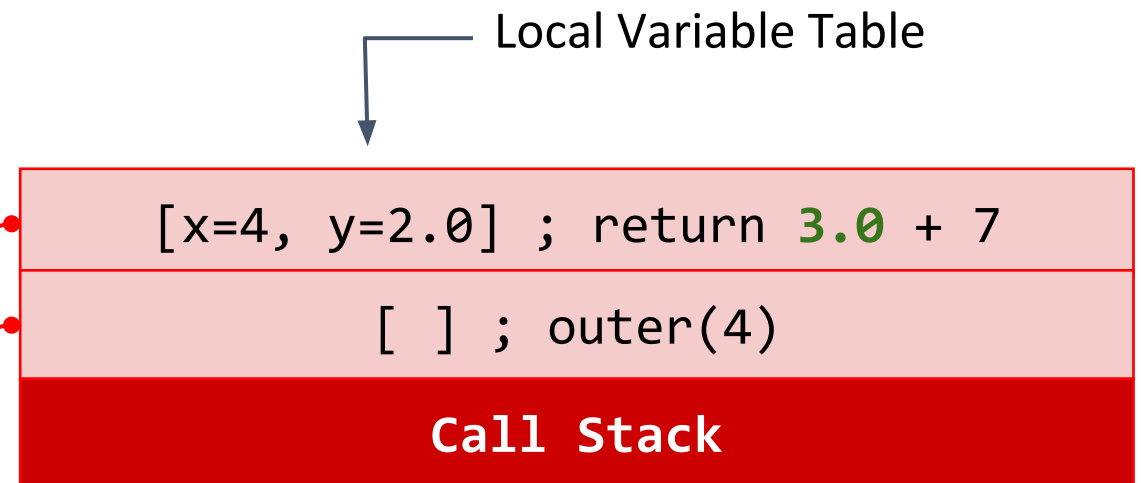
Call Stack

Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
outer(4)
```



Example: Function Calls on the Call Stack

```
def inner(x):  
    y = x + 1  
    return y
```

```
def outer(x):  
    y = x / 2  
    return inner(y) + 7
```

```
outer(4)
```



[] ; 10.0

Call Stack

Call Stack: Returned Values vs. Side Effects

Thinking in terms of the Call Stack can also help clarify the difference between returned values and side effects.

A **side effect** is something that happens within a single layer of the Call Stack that changes the program's **state**. A **returned value** is passed from the top layer on the Call Stack to the layer right below it.

Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

Interpreter:

```
>>>
```



[] ; wallet = 20.00

A call stack diagram consisting of two stacked rectangular boxes. The top box is light red and contains the text "[] ; wallet = 20.00". The bottom box is dark red and contains the text "Call Stack" in white. A red arrow points from the left side of the top box to the value "20.00" in the code snippet above.

Call Stack

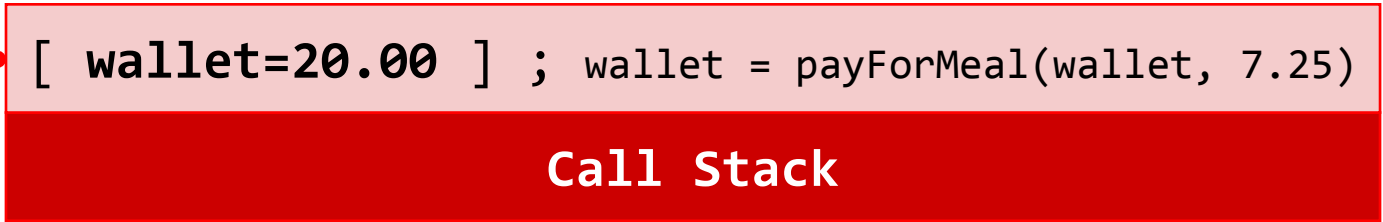
Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

Interpreter:

```
>>>
```



[wallet=20.00] ; wallet = payForMeal(wallet, 7.25)

The call stack is represented by a red-bordered box with a red background. The top part of the box contains the text "[wallet=20.00] ; wallet = payForMeal(wallet, 7.25)". A red arrow points from the top-left corner of this box to the second line of code in the previous block, "wallet = payForMeal(wallet, 7.25)".

Call Stack

Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

Interpreter:

```
>>>
```

[cash=20.00, cost=7.25] ; cash = cash-cost

[wallet=20.00] ; wallet = payForMeal(20.00, 7.25)

Call Stack

Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

```
Interpreter:  
Thanks!  
>>>
```

[cash=12.75, cost=7.25] ; print("Thanks!")

[wallet=20.00] ; wallet = payForMeal(20.00, 7.25)

Call Stack

Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

```
Interpreter:  
Thanks!  
>>>
```

[cash=12.75, cost=7.25] ; return cash

[wallet=20.00] ; wallet = payForMeal(20.00, 7.25)

Call Stack

Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

```
Interpreter:  
Thanks!  
>>>
```



[wallet=20.00] ; wallet = 12.75

The diagram shows a call stack frame. The top part is a light red box containing the text "[wallet=20.00] ; wallet = 12.75". A red arrow points from the top-left corner of this box to the argument "wallet" in the function call "payForMeal(wallet, 7.25)" in the code block above. The bottom part of the box is a dark red box containing the text "Call Stack".

Call Stack

Example: Returned Value vs. Side Effect

```
def payForMeal(cash, cost):  
    cash = cash - cost  
    print("Thanks!")  
    return cash
```

```
wallet = 20.00  
wallet = payForMeal(wallet, 7.25)
```

```
Interpreter:  
Thanks!  
>>>
```

[wallet=12.75]

Call Stack

Python Errors

Tokenizing and Parsing Errors are Syntax Errors

The first two steps of the Python translation process – 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 provided 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/quotes
```


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.

Examples of Logical Errors

```
print("2 + 2 = ", 5) # no error message, but wrong!
```

```
def double(x):  
    return x + 2 # adding instead of multiplying
```

Later, we'll use **assert statements** to catch logical errors in homework assignments.

Activity: Predict the Error Type

Let's test your knowledge of error types with a Kahoot!

Given a line of code, predict whether it will result in a Syntax Error, Runtime Error, Logical Error, or no error.

Join at kahoot.it

Asynchronous Kahoot:

https://kahoot.it/challenge/07628495?challenge-id=a750a494-3baa-4c36-81d2-898b6309e430_1599860660888

Learning Objectives

- Recognize that the process of **tokenizing, parsing, and translating** converts Python code into instructions a computer can execute
- Interpret and trace basic **bytecode** instructions
- Recognize how the different types of **errors** are raised at different points in the Python translation process
- **Feedback:** <https://bit.ly/110-feedback>