# Functions II & How Python Works

15-110 – Wednesday 02/10

# Announcements

- Feedback is now released for Check1
  - To view your feedback, open your assignment in Gradescope, then click on the question name on the right sidebar that you want to see feedback for.
  - Note that *all* rubric items are displayed by default; the rubric items applied to *your* submission should be highlighted.
  - If you find a grading error, use the Request Regrade button to ask the Lead TAs to take a second look

# Learning Objectives

- Trace the **call stack** to understand how Python keeps track of nested function calls
- Use **libraries** to import functions in categories like math, randomness, and graphics
- Recognize that the process of **tokenizing, parsing,** and **translating** converts Python code into instructions a computer can execute
- Recognize how the different types of **errors** are raised at different points in the Python translation process

# The Function Call Stack

# Function Calls Follow Order of Operations

Function calls evaluate to a single returned value; that means they are **expressions**. Therefore, we can **nest** function calls inside other expressions the same way we nest basic values and operations.

```
print(round(pow(abs(-12), 1/2), 2))
```

Just like in math, functions follow order of operations using parentheses. Start by evaluating the inner-most expressions, `abs(-12)` and `1/2`. Then evaluate the call to `pow`; then evaluate the call to `round`. Finally, evaluate the call to `print`.

# Function Calls in Function Definitions

Order of operations gets trickier to track when we write code in a function definition that **calls another function**.

When the code to the right calls the function `outer`, `outer` will run a bit of code, then call the function `inner`.

Python needs to keep track of which variables are in scope at any given point, and where returned values should be sent. It does this with a **call stack**.

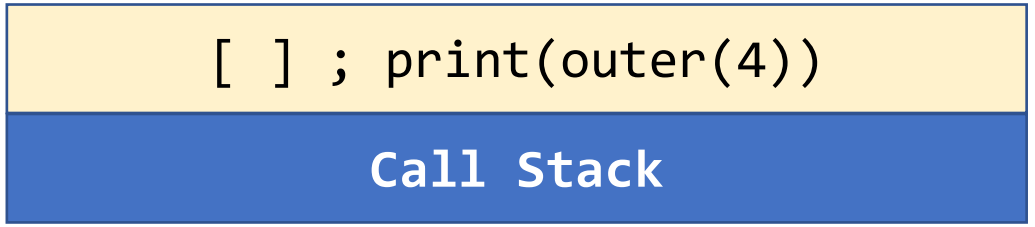
```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

# Functions and the Call Stack

**Interpreter:**
>>>

Python executes the function line-by-line until it reaches a function call. It saves the **global state** of the program on the call stack.

Then it just adds a new layer to the stack, on the top!



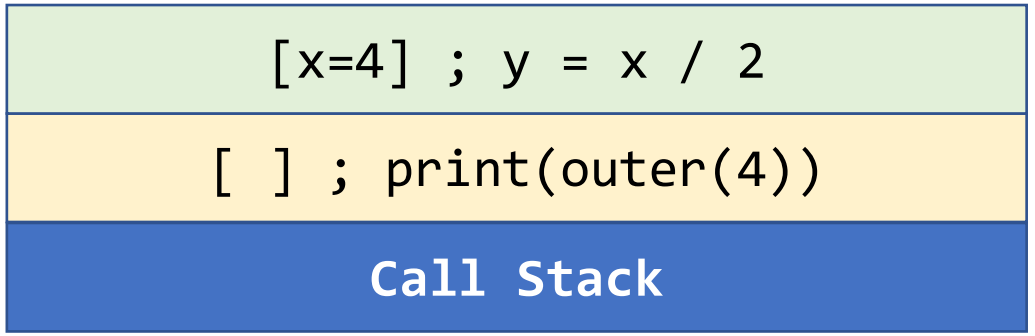

```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

# Functions and the Call Stack

**Interpreter:**
>>>

Python moves through the definition line by line, using the state of the **top** stack level when it needs to look up variable values. It can also update that top-stack-level state as needed.

The lower levels are ignored for now.



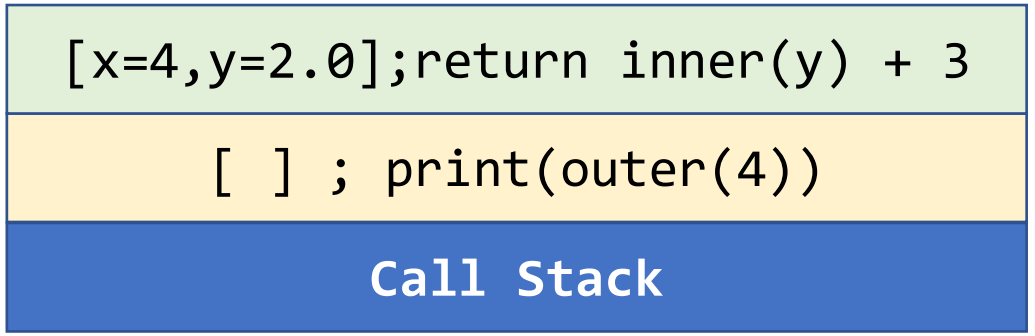

```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

# Functions and the Call Stack

**Interpreter:**
>>>

When Python reaches the call to inner inside of outer, it just adds another level to the stack!

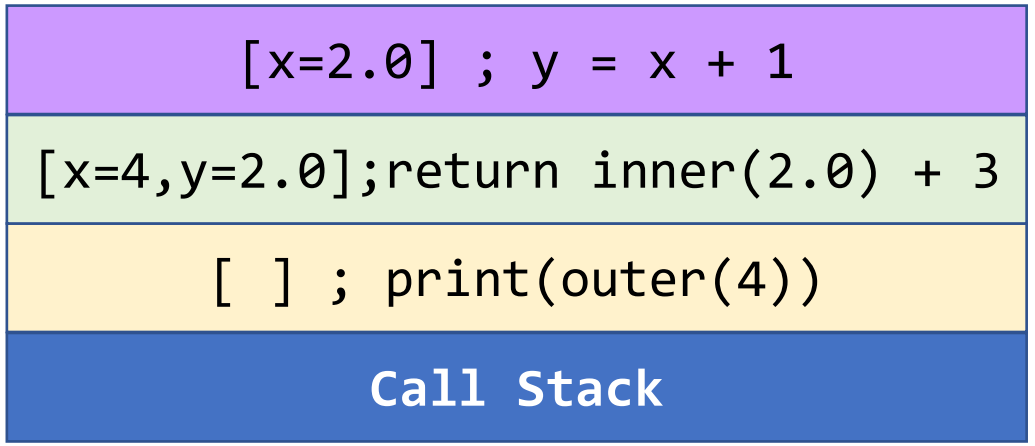
```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```



# Functions and the Call Stack

**Interpreter:**
>>>

Again, Python will move through the function normally.

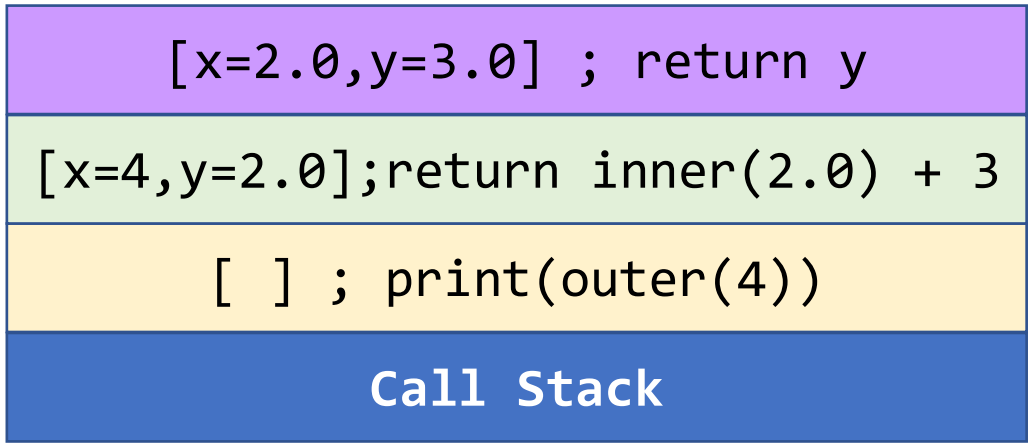



```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

# Functions and the Call Stack

**Interpreter:**
>>>

When it reaches a **return statement**, it will evaluate the returned value, then **remove the top level of the stack**. The returned value is then sent to the call on the new top level, where it replaces the call.

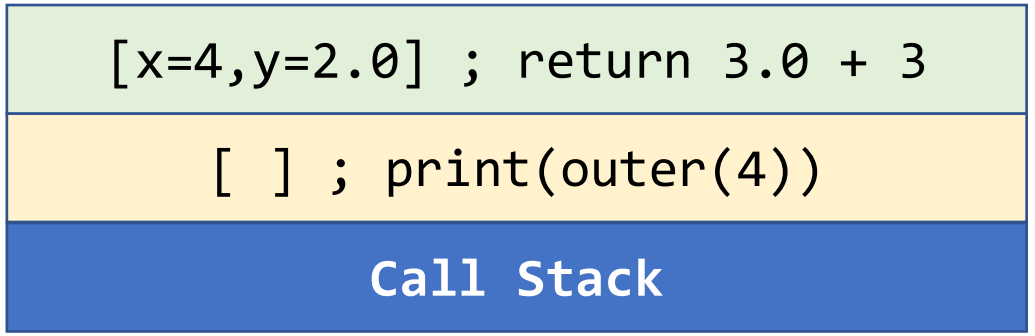



```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

# Functions and the Call Stack

**Interpreter:**
>>>

When it reaches a **return statement**, it will evaluate the returned value, then **remove the top level of the stack**. The returned value is then sent to the call on the new top level, where it replaces the call.



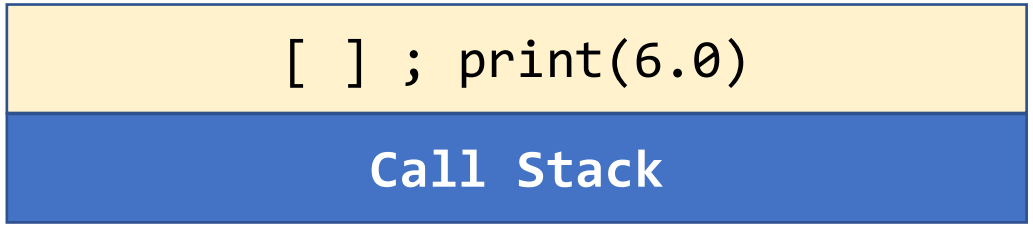

```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

# Functions and the Call Stack

**Interpreter:**
>>>

The `print` call will also technically put a new level on the call stack, but we can't see into what the `print` function is doing since we didn't write it ourselves.

```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

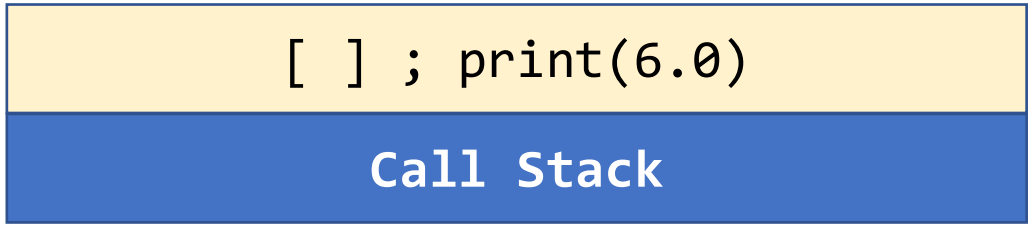


# Functions and the Call Stack

**Interpreter:**
```
>>> 6.0
```

The `print` call causes a side effect...

```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```

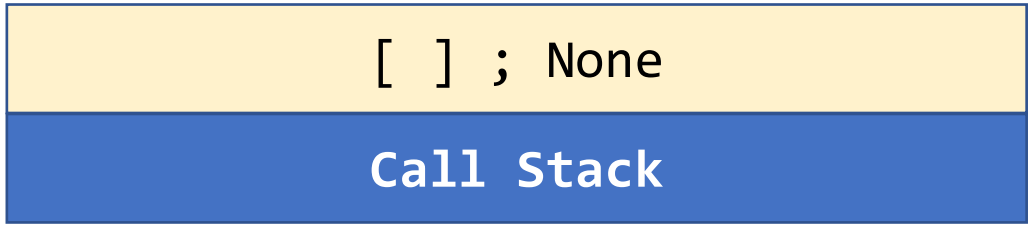


# Functions and the Call Stack

**Interpreter:**
```
>>> 6.0
```

... then evaluates to None. Then it's done!

```
def outer(x):
    y = x / 2
    return inner(y) + 3

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

print(outer(4))
```



# Activity: Trace the Function Calls

Now you try it! Given the code to the right, draw a call stack to determine what will be printed.

Make sure to keep track of the state on each level.

```
def calculateTip(cost):
    tipRate = 0.2
    return cost * tipRate

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

wallet = 20.00
wallet = payForMeal(wallet, 8.00)
print("Money remaining:", wallet)
```

# Libraries

# Import Adds Code from Libraries

The Python language has a ton of pre-built functions, but most aren't included in the built-in package (the one available by default). Most of the functions are organized into separate **libraries**.

To use a function from a library, you must **import** the library. This makes it possible to access the functions and variables in that collection. You can do this with the code:

```
import libraryName
```

All the Python libraries have **documentation** online that describes which functions are available and what they do. Find it by searching docs.python.org/3/ . It's better to check the documentation as needed than to try to memorize library functions.

# Importing the math Library

For example, we can import the **math** library to add more mathematical capabilities. Note that we must put `math.` in front of each function or variable name we use, to specify it came from that library.

```
import math
math.ceil(6.5) # ceiling of a float number
math.log(64, 2) # finds the log of 64 with base 2
math.radians(90) # converts degrees to radians
math.pi # it's π!
```

# Importing the random library

Importing libraries lets us get more creative with programming. For example, the **random** library lets us generate random numbers, which can help produce novel behavior.

```
import random
random.randint(1, 10) # picks a random int between 1-10 inclusive
random.random() # picks a random float between 0-1
```

# Importing a graphics library

Finally, to get really creative, we can produce graphics with programming! We'll do this with the **tkinter** library, which makes it possible to draw shapes on a separate screen.

```
import tkinter
```

# Tkinter Starter Code

We need to run some code before and after our graphics code to make it work.

The `root` is the window. The `canvas` is the thing on the window where we can draw shapes.

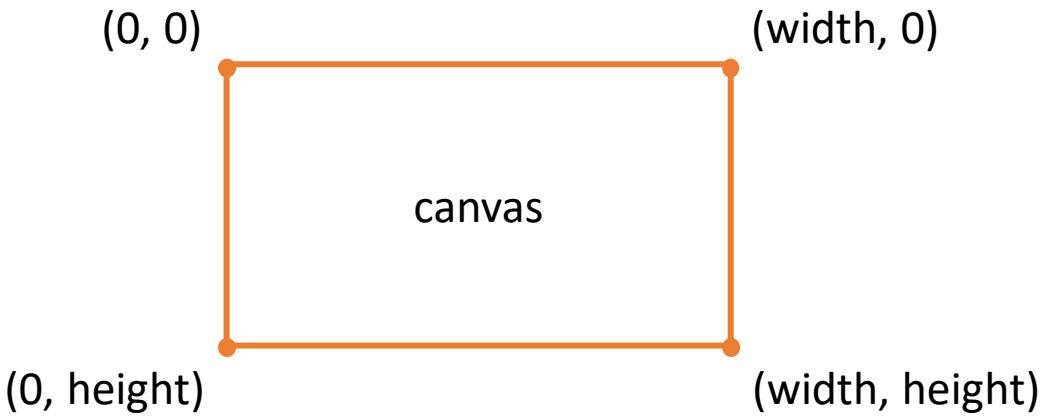
The `root.mainloop()` line will tell the window to stay open until we press the X button.

```
import tkinter as tk # shorten library name

# You write code in here!
def draw(canvas):
    pass

def makeCanvas(w, h):
    root = tk.Tk()
    canvas = tk.Canvas(root, width=w,
                       height=h)
    canvas.configure(bd=0,
                     highlightthickness=0)
    canvas.pack()

    draw(canvas) # call your code here

    root.mainloop()

makeCanvas(400, 400)
```

# Coordinates on the Canvas Grow Down-Right

The **canvas** created by the starter code is the thing we'll draw graphics on. It's a two-dimensional grid of pixels. This grid has a pre-set **width** and **height;** the number of pixels from left to right and the number of pixels from top to bottom.

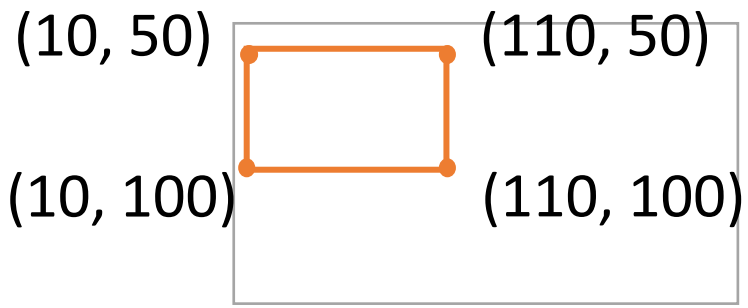
We can refer to pixels on the canvas by their (x, y) coordinates. However, these coordinates are different from coordinates on mathematical graphs – the origin starts at the **top left corner** of the canvas.



# Drawing a Rectangle

To draw a rectangle, use the function `canvas.create_rectangle`. This function takes four required arguments: the x and y coordinates of the **left-top** corner, and the x and y coordinates of the **right-bottom** corner. The rectangle will then be drawn between those two points.

```
canvas.create_rectangle(10, 50, 110, 100)
```



# Graphics – Side Effects and Returned Values

When the rectangle is drawn on the canvas, we can't use it in future computations. That's a **side effect**.

The graphics function call also returns something – an integer ID associated with the drawn shape. We won't use that value in this class.

You can draw a lot more than just rectangles with Tkinter graphics! Check out the bonus slides on graphics to find more shapes.

# Keyword Arguments Add Variety

With the basic parameters, we can only draw outlines of shapes. By adding **keyword arguments**, we can change the properties of these shapes.

A keyword argument is an argument is associated with a specific name instead of a position in the function call. We can put keyword arguments in any order we like as long as they occur after the main arguments.

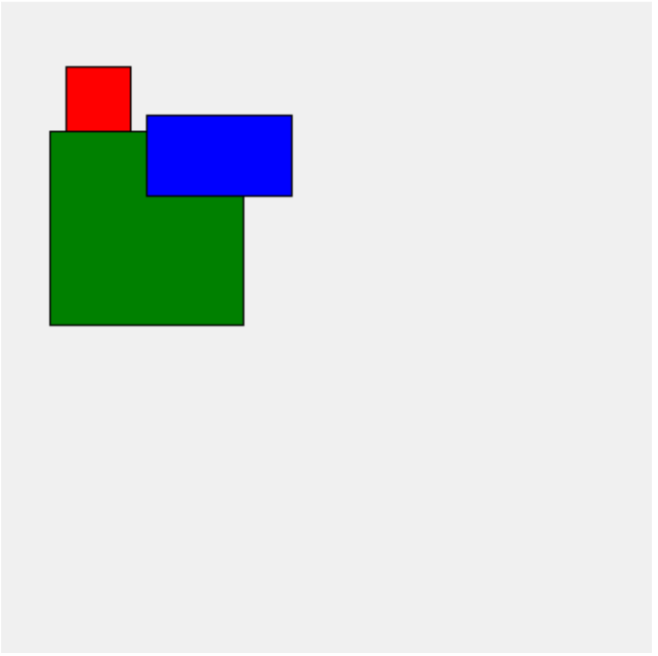
Keyword arguments can have **default values**, which is why we don't need to include them in every graphics call. To change that default value, include the keyword, followed by =, followed by the new value in the function call.

```
canvas.create_rectangle(50, 100, 150, 200, fill="green")
```

# Keyword Argument - fill

The `fill` argument can be used on any shape. It uses a string (the name of the color) to change the color of the shape.

```
canvas.create_rectangle(40, 40, 80, 140, fill="red")

canvas.create_rectangle(30, 80, 150, 200, fill="green")

canvas.create_rectangle(90, 70, 180, 120, fill="blue")
```



Note that when we draw shapes on top of each other, the one on top is the **last one called**. Order matters!

Check the bonus slides to find more keyword arguments.

# 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 the text directly.

A computer *does* know how to follow a small set of instructions that are built into its hardware. These instructions are called machine code. One step up from machine code is **bytecode**, a language for a slightly-simplified computer.

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 = 52
col = x / (500/50)
```

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**, **=**, **52**, **newline**, **col**, **=**, **x**, **/**, **(**, **500**, **/**, **50**, **)**

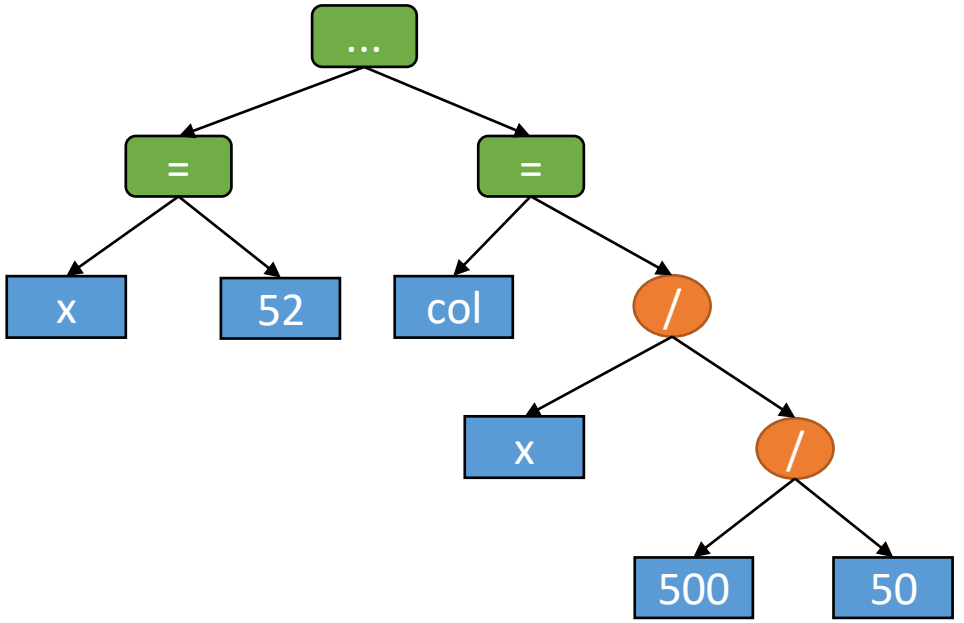
# Parsing Groups Tokens by Task

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 **52** as the value.

```
x, =, 52, newline,
col, =, x, /, (, 500, /, 50, )
```



# Translate 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 into every computer's hardware.

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

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

# 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_NAME 0 # load the variable at position 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 a stack, which keeps track of the program's **state** as it runs.

When we run a Python program, we're actually running bytecode behind the scenes!

# 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 provide does not follow the rules of the Python language's syntax.

# Examples of Syntax Errors

Most syntax errors are called **SyntaxError**, 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 in the next lecture.

# 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

# Learning Objectives

- Trace the **call stack** to understand how Python keeps track of nested function calls
- Use **libraries** to import functions in categories like math, randomness, and graphics
- Recognize that the process of **tokenizing, parsing,** and **translating** converts Python code into instructions a computer can execute
- Recognize how the different types of **errors** are raised at different points in the Python translation process
- Feedback: http://bit.ly/110-s21-feedback