Python 102: Beyond the Basics

PyTexas 2024 - April 19, 2024

Mason Egger

Hi. I'm Mason!

- Sr. Technical Curriculum Developer
- Programmer (http://github.com/masonegger)
- Author (https://mason.dev)
- PyTexas Community Organizer (https://pytexas.org)
- Teacher

Logistics

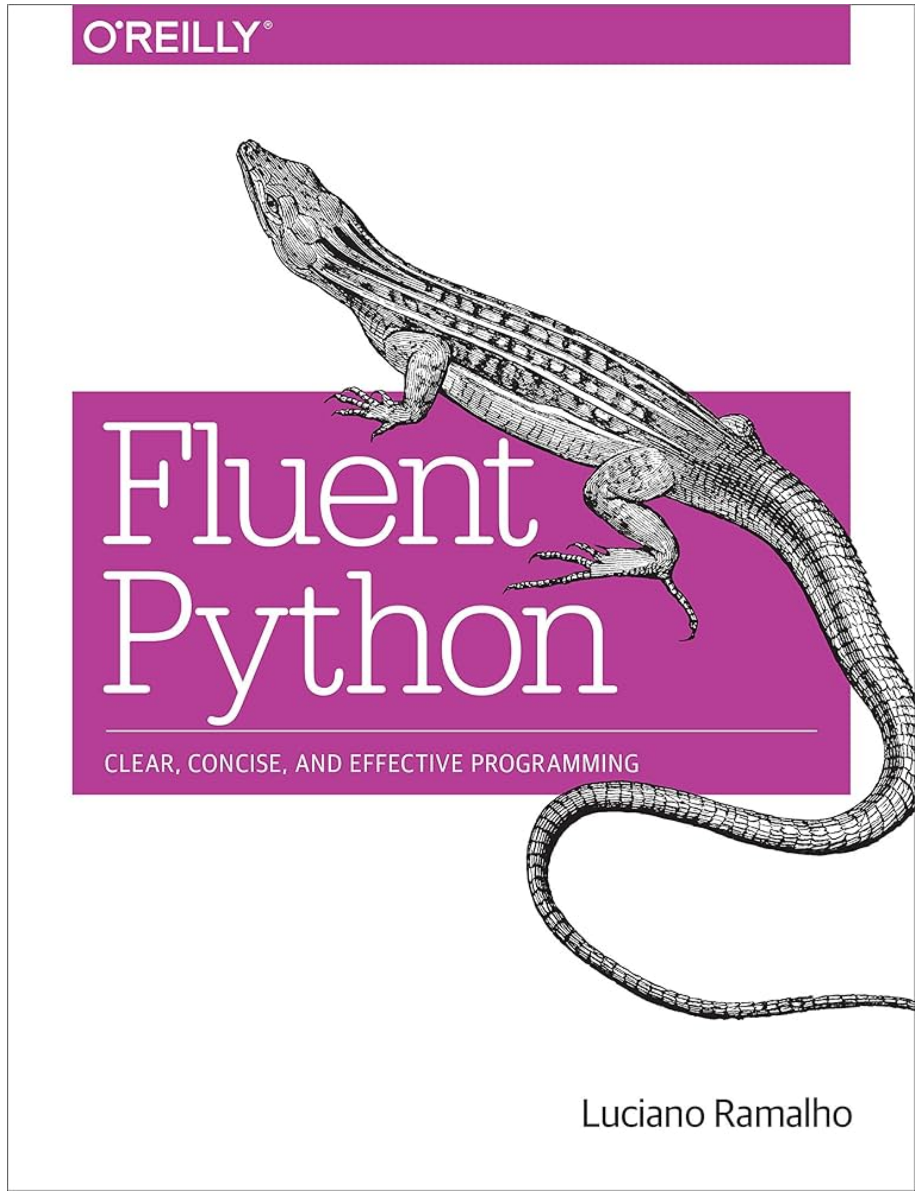
- Schedule
- Asking questions
- Getting help with the exercises

Exercise Environment

- I provide a development environment for you in this workshop
  - It uses Google Colaboratory, a hosted Jupyter Notebook Service
  - You access it through your browser (may require you to log in to Google)
    - You may also clone the exercises onto your local machine
- I will now demonstrate how to access and use it

bit.ly/pytexas2024-102

Inspiration for the Course



Today...

- Evaluate what makes a Python function **first-class**
- Construct **decorators** to enhance the behavior of a function
- Implement **comprehensions** to simplify the construction of new sequences
- Extend objects using **dunder methods**, allowing for interaction with built-in functions and operators
- Use **context managers** to manage the spin-up/tear-down of various objects and processes

Part I: *First-Class Functions*

First-Class Functions

- In Python, all functions are considered *First-Class Functions*
- Functions are considered *First-Class* if they can be:
  - Created at runtime
  - Assigned to a variable or element in a data structure
  - Passed as an argument to a function
  - Returned as the result of a function
- **"First-Class Functions" means being able to treat a function like an object**

Functions as Objects (Variable Assignment)

You're familiar with creating objects in Python

```
text = "Hello PyTexas"
print(text)
```

Functions as Objects (Function Definition)

And you're familiar with creating functions in Python

```
# function definition
def my_func(text):
  print(text)

# function call
my_func("Hello PyTexas from a function")
```

Functions as Objects (Function Definition)

And you're familiar with creating functions in Python

```
# function definition
def my_func(text):
  print(text)

# function call
my_func("Hello PyTexas from a function")
```

Functions as Objects (Function Definition)

And you're familiar with creating functions in Python

```
# function definition
def my_func(text):
  print(text)

# function call
my_func("Hello PyTexas from a function")
```

Assigning a Function to a Variable

But you can also assign a function to a variable

```
# function definition
def my_func(text):
  print(text)

# function assignment
x = my_func

# function call
x("Hello PyTexas from a function assigned to a variable")
```

Assigning a Function to a Variable

But you can also assign a function to a variable

```
# function definition
def my_func(text):
  print(text)

# function assignment
x = my_func

# function call
x("Hello PyTexas from a function assigned to a variable")
```

Assigning a Function to a Variable

But you can also assign a function to a variable

```
# function definition
def my_func(text):
  print(text)

# function assignment
x = my_func

# function call
x("Hello PyTexas from a function assigned to a variable")
```

Assigning a Function to a Variable

But you can also assign a function to a variable

```
# function definition
def my_func(text):
  print(text)

# function assignment
x = my_func

# function call
x("Hello PyTexas from a function assigned to a variable")
```

Passing Functions to Other Functions

You can even pass functions as arguments

```
# function definition
def pass_func(func):
  func("Passing a function as a parameter")

# Using `my_func` from the previous slide/cell
pass_func(my_func)
```

Passing Functions to Other Functions

You can even pass functions as arguments

```
# function definition
def pass_func(func):
  func("Passing a function as a parameter")

# Using `my_func` from the previous slide/cell
pass_func(my_func)
```

Passing Functions to Other Functions

You can even pass functions as arguments

```
# function definition
def pass_func(func):
  func("Passing a function as a parameter")

# Using `my_func` from the previous slide/cell
pass_func(my_func)
```

Passing Functions to Other Functions

You can even pass functions as arguments

```
# function definition
def pass_func(func):
  func("Passing a function as a parameter")

# Using `my_func` from the previous slide/cell
pass_func(my_func)
```

Functions are Objects

This is possible because functions in Python are objects

```
print(my_func)
print(pass_func)
```

Other Object Properties

Since a function is just an object in Python, you can use the function the same way you would use any object. You can:

- Pass them to other functions
- Return functions from other functions
- Store functions in data structures

Storing a Function in a Dict

```
my_dict = {"my_func": my_func}
my_dict["my_func"]("Hello PyTexas from a dict")
```

Storing a Function in a Dict

```
my_dict = {"my_func": my_func}
my_dict["my_func"]("Hello PyTexas from a dict")
```

Storing a Function in a Dict

```
my_dict = {"my_func": my_func}
my_dict["my_func"]("Hello PyTexas from a dict")
```

Higher-Order Functions

A function that takes a function as an argument or returns a functions as the result is considered a *higher-order function*. Higher-Order Functions are great for abstracting and modularizing code, allowing you to compose more complex logic out of simpler functions.

```
# function definition
def pass_func(func):
  func("Passing a function as a parameter")

# Using `my_func` from the previous slide/cell
pass_func(my_func)
```

Higher-Order Functions in the Standard Library

The standard library is filled with higher-order functions. Popular ones include `map`, `filter`, `reduce`, and `sort`.

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]
print(len("bbq"))

# The `len` function, which determines the length of a
# str was passed to the `sorted` function
sorted(my_list, key=len)
```

Higher-Order Functions in the Standard Library

The standard library is filled with higher-order functions. Popular ones include `map`, `filter`, `reduce`, and `sort`.

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]
print(len("bbq"))

# The `len` function, which determines the length of a
# str was passed to the `sorted` function
sorted(my_list, key=len)
```

Higher-Order Functions in the Standard Library

The standard library is filled with higher-order functions. Popular ones include `map`, `filter`, `reduce`, and `sort`.

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]
print(len("bbq"))

# The `len` function, which determines the length of a
# str was passed to the `sorted` function
sorted(my_list, key=len)
```

Higher-Order Functions in the Standard Library

The standard library is filled with higher-order functions. Popular ones include `map`, `filter`, `reduce`, and `sort`.

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]
print(len("bbq"))

# The `len` function, which determines the length of a
# str was passed to the `sorted` function
sorted(my_list, key=len)
```

Passing `args` and `kwargs` to a Function

When defining a function, you can pass either a specific set of arguments or an undefined amount using `*args` for positional arguments or `**kwargs` for keyword arguments.

```
def my_func(*args, **kwargs):
    print(args)
    print(kwargs)

my_func("hello", "goodbye", language="en", capitalize=True)
```

Passing `args` and `kwargs` to a Function

When defining a function, you can pass either a specific set of arguments or an undefined amount using `*args` for positional arguments or `**kwargs` for keyword arguments.

```
def my_func(*args, **kwargs):
    print(args)
    print(kwargs)

my_func("hello", "goodbye", language="en", capitalize=True)
```

Passing `args` and `kwargs` to a Function

When defining a function, you can pass either a specific set of arguments or an undefined amount using `*args` for positional arguments or `**kwargs` for keyword arguments.

```
def my_func(*args, **kwargs):
    print(args)
    print(kwargs)

my_func("hello", "goodbye", language="en", capitalize=True)
```

Passing `args` and `kwargs` to a Function

When defining a function, you can pass either a specific set of arguments or an undefined amount using `*args` for positional arguments or `**kwargs` for keyword arguments.

```
def my_func(*args, **kwargs):
    print(args)
    print(kwargs)

my_func("hello", "goodbye", language="en", capitalize=True)
```

Passing `args` and `kwargs` to a Function

When defining a function, you can pass either a specific set of arguments or an undefined amount using `*args` for positional arguments or `**kwargs` for keyword arguments.

```
def my_func(*args, **kwargs):
    print(args)
    print(kwargs)

my_func("hello", "goodbye", language="en", capitalize=True)
```

Passing `args` and `kwargs` to a Function

When defining a function, you can pass either a specific set of arguments or an undefined amount using `*args` for positional arguments or `**kwargs` for keyword arguments.

```
def my_func(*args, **kwargs):
    print(args)
    print(kwargs)

my_func("hello", "goodbye", language="en", capitalize=True)
```

Anonymous Functions (aka Lambda Functions)

- Anonymous functions can take any number of arguments, but can only have one expression
- A concise way of creating small, one-line functions
- Useful where a short function is needed for a specific purpose, such as passing a simple function as an arguement to another function
- Implemented using the `lambda` keyword in Python

Anonymous Functions Examples

Example 1

```
add_one = lambda x: x+1
add_one(2)
```

Example 2

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]

# Sort by the last letter
sorted(my_list, key=lambda x: x[-1])
```

Anonymous Functions Examples

Example 1

```
add_one = lambda x: x+1
add_one(2)
```

Example 2

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]

# Sort by the last letter
sorted(my_list, key=lambda x: x[-1])
```

Anonymous Functions Examples

Example 1

```
add_one = lambda x: x+1
add_one(2)
```

Example 2

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]

# Sort by the last letter
sorted(my_list, key=lambda x: x[-1])
```

Anonymous Functions Examples

Example 1

```
add_one = lambda x: x+1
add_one(2)
```

Example 2

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]

# Sort by the last letter
sorted(my_list, key=lambda x: x[-1])
```

Anonymous Functions Examples

Example 1

```
add_one = lambda x: x+1
add_one(2)
```

Example 2

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]

# Sort by the last letter
sorted(my_list, key=lambda x: x[-1])
```

Anonymous Functions Examples

Example 1

```
add_one = lambda x: x+1
add_one(2)
```

Example 2

```
my_list = ["bluebonnet", "lonestar", "armadillo", "bbq"]

# Sort by the last letter
sorted(my_list, key=lambda x: x[-1])
```

Function Introspection

Functions have many attributes. Use the `dir` function to view all the methods associated with the function.

```
dir(my_func)
```

Using `dir` with Classes

You can use `dir` to see the methods within a class

```
dir(list)
```

Using `help()` to introspect

You can also use the `help()` command to read a function or classes documentation.

```
help(list)
```

Summary (Pt. 1)

- Functions are considered First-Class in Python
- First-Class means that the function is treated like an object
- Just like other objects, functions can be:
  - Created at runtime
  - Assigned to a variable or element in a data structure
  - Passed as an argument to a function
  - Returned as the result of a function

Summary (Pt. 2)

- Functions that take other functions as parameters, or return a function as a result is known as a *Higher-Order Function*
- Anonymous functions are implemented using the `lambda` keyword, and are good for creating concise, one off functions.

Exercise 1 - First-Class Functions

- In these exercises you will:
  - Implement a Higher-Order function
  - Implement a lambda function being passed to `filter` and `sorted`
- Go to the Exercise Directory in the Google Drive and open the Practice Directory
- Open *01-First-Class-Functions-Exercises-Solution.ipynb* and follow the instructions
- If you get stuck, raise your hand and someone will come by and help. You can also check the `Solution` directory for the answers
- **You have 10 mins**

Part II: *Decorators and Closures*

Decorators & Closures

- Decorators allow us to "mark" functions to enhance their behavior
- Work by wrapping another function and adding functionality to it
- Can be applied to both functions and classes
- Allow for reusability and promote a clean, concise coding style

Decorators in the Wild

You may have seen decorators before:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def helloworld():
    return "Hello World!"
```

Decorators in the Wild

You may have seen decorators before:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def helloworld():
    return "Hello World!"
```

Decorator Syntax

A decorator is the name of the decorator, prepended with the `@` sign, place above the function definition

```
@my_decorator
def my_func(text):
  print(text)
```

Here we say that `my_decorator` decorates `my_func`

Decorator Syntax

A decorator is the name of the decorator, prepended with the @ sign, place above the function definition

```
@my_decorator
def my_func(text):
  print(text)
```

Here we say that `my_decorator` decorates `my_func`

A Decorator is Higher-Order Function

Decorators are syntactic sugar for Higher-Order Functions. These two snippets of code are equivalent.

```
@my_decorator
def my_function(text):
  print(text)
```

```
# function definition
def my_function(text):
    print(text)

# function call
my_function = my_decorator(text)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
    def inner():
        return "Running inner"
    return inner

@decorator
def my_func():
    return "Hello PyTexas"

result = my_func()
print(result)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Decorator Example

A decorator usually replaces a function with a different one.

```
def decorator(func):
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Passing Variables to the Decorator

When passing variables to the decorator function, it's easy to forget to include the variables in the new function definition. Remember that the function object will be replaced by the decorator. So any parameters required in the original function header will be lost.

```
def decorator(func):
  def inner(text):
    return f"Passed message: {text}"
  return inner

@decorator
def my_func():
  return "Hello to PyTexas"

result = my_func("Hello from PyTexas")
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

Using Decorators to Enhance Capabilities

However, it seems odd to just throw the entire function away. Decorators are usually used to add functionality to functions.

```
def reverse(func):
  def inner():
    x = func()
    return x[::-1]
  return inner

@reverse
def my_func():
  return "Hello PyTexas"

result = my_func()
print(result)
```

When are Decorators Run?

Decorators a run right after the decorated function is defined. This usually happens at *import time*, i.e., when a module is loaded by Python.

```
import time

def decorator(func):
  print("Decorator being run")
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

time.sleep(5)

print(my_func())
```

When are Decorators Run?

Decorators a run right after the decorated function is defined. This usually happens at *import time*, i.e., when a module is loaded by Python.

```
import time

def decorator(func):
  print("Decorator being run")
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

time.sleep(5)

print(my_func())
```

When are Decorators Run?

Decorators a run right after the decorated function is defined. This usually happens at *import time*, i.e., when a module is loaded by Python.

```
import time

def decorator(func):
  print("Decorator being run")
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

time.sleep(5)

print(my_func())
```

When are Decorators Run?

Decorators a run right after the decorated function is defined. This usually happens at *import time*, i.e., when a module is loaded by Python.

```
import time

def decorator(func):
  print("Decorator being run")
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

time.sleep(5)

print(my_func())
```

When are Decorators Run?

Decorators a run right after the decorated function is defined. This usually happens at *import time*, i.e., when a module is loaded by Python.

```
import time

def decorator(func):
  print("Decorator being run")
  def inner():
    return "Running inner"
  return inner

@decorator
def my_func():
  return "Hello PyTexas"

time.sleep(5)

print(my_func())
```

And now for something completely different....

Variable Scoping Review

In order to fully understand closures, we need to take a step back and review how scoping is handled in Python. Does this code run?

```
def f1(a):
  print(a)
  print(b)

f1(1)
```

Variable Scoping Review

In order to fully understand closures, we need to take a step back and review how scoping is handled in Python. Does this code run?

```
def f1(a):
  print(a)
  print(b)

f1(1)
```

Variable Scoping Review Cont.

The variable b in this instance is known as a *free* variable, meaning it is not bound to the local scope

```
b = 6
def f2(a):
  print(a)
  print(b)

f2(1)
```

Variable Scoping Review Cont.

The variable b in this instance is known as a *free* variable, meaning it is not bound to the local scope

```
b = 6
def f2(a):
  print(a)
  print(b)

f2(1)
```

Variable Scoping Review Cont.

The variable b in this instance is known as a *free* variable, meaning it is not bound to the local scope

```
b = 6
def f2(a):
  print(a)
  print(b)

f2(1)
```

Variable Scoping Review Cont.

The variable b in this instance is known as a *free* variable, meaning it is not bound to the local scope

```
b = 6
def f2(a):
  print(a)
  print(b)

f2(1)
```

Variable Scoping Review Cont.

What about this?

```
d = 6
def f3(c):
  print(c)
  print(d)
  d = 8

f3(1)
```

Wait, what happened?

By assigning a value to `d` within the function, it was no longer considered a `free` variable, but a local variable within the scope of `f3`. This ignored the external declaration of `d`.

This is a design choice by Python, not a bug. It is designed to prevent accidental mutation of global variables.

Global Variables

One way to fix this, use the `global` keyword to tell Python that the variable is in face global.

```
f = 6
def f4(e):
  global f
  print(e)
  print(f)
  f = 8

f4(1)
print(f)
```

Global Variables

One way to fix this, use the `global` keyword to tell Python that the variable is in face global.

```
f = 6
def f4(e):
  global f
  print(e)
  print(f)
  f = 8

f4(1)
print(f)
```

And back to your regularly scheduled content!

Closures

A *closure* is a function with an extended scope that encompasses non-global variables referenced in the body of the function but not defined there.

For example: How would you implement a function that has the following output?

```
>>> sum(1)
1
>>> sum(1)
2
>>> sum(4)
6
```

Your first thought may be to use a global variable, but global variables are often not best practice here. This is where we use closures.

Implementing a Closure

Will this work?

```
def calc_sum():
  total = 0

  def add_num(num):
    total += num
    return total

  return add_num

sum = calc_sum()
sum(1)
sum(1)
sum(4)
```

The Closure Area

The area within the first function but external to the second function is known as the closure.

```
def calc_sum():
  # BEGIN CLOSURE {
  total = 0
  # } END CLOSURE
  def add_num(num):
    total += num
    return total

  return add_num
```

The Closure Area

The area within the first function but external to the second function is known as the closure.

```
def calc_sum():
  # BEGIN CLOSURE {
  total = 0
  # } END CLOSURE
  def add_num(num):
    total += num
    return total

  return add_num
```

Implementing a Closure Cont.

The `nonlocal` keyword let's us tell Python that a variable is not local to the scope of the function, but should be allowed to be changed.

```
def calc_sum():
  total = 0

  def add_num(num):
    nonlocal total
    total += num
    return total

  return add_num

sum = calc_sum()
sum(1)
sum(1)
sum(4)
```

Implementing a Closure Cont.

The `nonlocal` keyword let's us tell Python that a variable is not local to the scope of the function, but should be allowed to be changed.

```
def calc_sum():
  total = 0

  def add_num(num):
    nonlocal total
    total += num
    return total

  return add_num

sum = calc_sum()
sum(1)
sum(1)
sum(4)
```

Using Closures with Decorators

Now you can use closures to maintain state in-between decorator calls.

```
def count_calls(func):
  total = 0

  def count_invoke(name):
    nonlocal total
    func(name)
    total += 1
    return total

  return count_invoke

@count_calls
def sell_tickets(name):
  print(f"Ticket sold to {name}")

sell_tickets("Laura")
sell_tickets("Pandy")
```

Chaining Decorators

- Decorators can be chained together
  - This means you can add more than one decorator to a function
- Decorators are applied from bottom to top

```
@make_h1_md
@make_bold_md
def greeting(text):
  return text
```

Chaining Decorators

- Decorators can be chained together
  - This means you can add more than one decorator to a function
- Decorators are applied from bottom to top

```
@make_h1_md
@make_bold_md
def greeting(text):
  return text
```

Chaining Decorators

- Decorators can be chained together
  - This means you can add more than one decorator to a function
- Decorators are applied from bottom to top

```
@make_h1_md
@make_bold_md
def greeting(text):
  return text
```

Chaining Decorators

- Decorators can be chained together
    - This means you can add more than one decorator to a function
- Decorators are applied from bottom to top

```
@make_h1_md
@make_bold_md
def greeting(text):
  return text
```

Chaining Decorators Example

```
def make_h1_md(func):
  def wrapper(text):
      return "# " + func(text)
  return wrapper

def make_bold_md(func):
  def wrapper(text):
      return "**" + func(text) + "**"
  return wrapper

@make_h1_md
@make_bold_md
def greeting(text):
  return text

print(greeting("hello"))
```

Passing Parameters to Decorators

- It is also possible to pass a parameter directy to the decorator.
    - In `Flask` you would apply the `app.route("/")` decorator to the function that will be served at route `/`.
- However, doing this requires wrapping your decorator in another function and calling that.

```
def decorator_with_argument(name):
  def decorator(func):
    def wrapper(text):
      return func(text) +  f" {name}"
    return wrapper
  return decorator

@decorator_with_argument("Mason")
def greeting(text):
  return text[0].upper() + text[1:]

greeting("hola")
```

Passing Parameters to Decorators

- It is also possible to pass a parameter directy to the decorator.
    - In `Flask` you would apply the `app.route("/")` decorator to the function that will be served at route `/`.
- However, doing this requires wrapping your decorator in another function and calling that.

```
def decorator_with_argument(name):
  def decorator(func):
    def wrapper(text):
      return func(text) +  f" {name}"
    return wrapper
  return decorator

@decorator_with_argument("Mason")
def greeting(text):
  return text[0].upper() + text[1:]

greeting("hola")
```

Passing Parameters to Decorators

- It is also possible to pass a parameter directy to the decorator.
  - In `Flask` you would apply the `app.route("/")` decorator to the function that will be served at route `/`.
- However, doing this requires wrapping your decorator in another function and calling that.

```
def decorator_with_argument(name):
  def decorator(func):
    def wrapper(text):
      return func(text) +  f" {name}"
    return wrapper
  return decorator

@decorator_with_argument("Mason")
def greeting(text):
  return text[0].upper() + text[1:]

greeting("hola")
```

Passing Parameters to Decorators

- It is also possible to pass a parameter directy to the decorator.
  - In `Flask` you would apply the `app.route("/")` decorator to the function that will be served at route `/`.
- However, doing this requires wrapping your decorator in another function and calling that.

```
def decorator_with_argument(name):
  def decorator(func):
    def wrapper(text):
      return func(text) +  f" {name}"
    return wrapper
  return decorator

@decorator_with_argument("Mason")
def greeting(text):
  return text[0].upper() + text[1:]

greeting("hola")
```

Passing Parameters to Decorators

- It is also possible to pass a parameter directy to the decorator.
    - In `Flask` you would apply the `app.route("/")` decorator to the function that will be served at route `/`.
- However, doing this requires wrapping your decorator in another function and calling that.

```
def decorator_with_argument(name):
  def decorator(func):
    def wrapper(text):
      return func(text) +  f" {name}"
    return wrapper
  return decorator

@decorator_with_argument("Mason")
def greeting(text):
  return text[0].upper() + text[1:]

greeting("hola")
```

Passing Parameters to Decorators

- It is also possible to pass a parameter directy to the decorator.
    - In `Flask` you would apply the `app.route("/")` decorator to the function that will be served at route `/`.
- However, doing this requires wrapping your decorator in another function and calling that.

```
def decorator_with_argument(name):
  def decorator(func):
    def wrapper(text):
      return func(text) +  f" {name}"
    return wrapper
  return decorator

@decorator_with_argument("Mason")
def greeting(text):
  return text[0].upper() + text[1:]

greeting("hola")
```

Summary (Pt. 1)

- Decorators allow us to "mark" functions to enhance their behavior
- Decorators are syntactic sugar for Higher-Order Functions
- Decorators return an entirely new function that may or may not call the original function
- Decorators are first run at *import time*

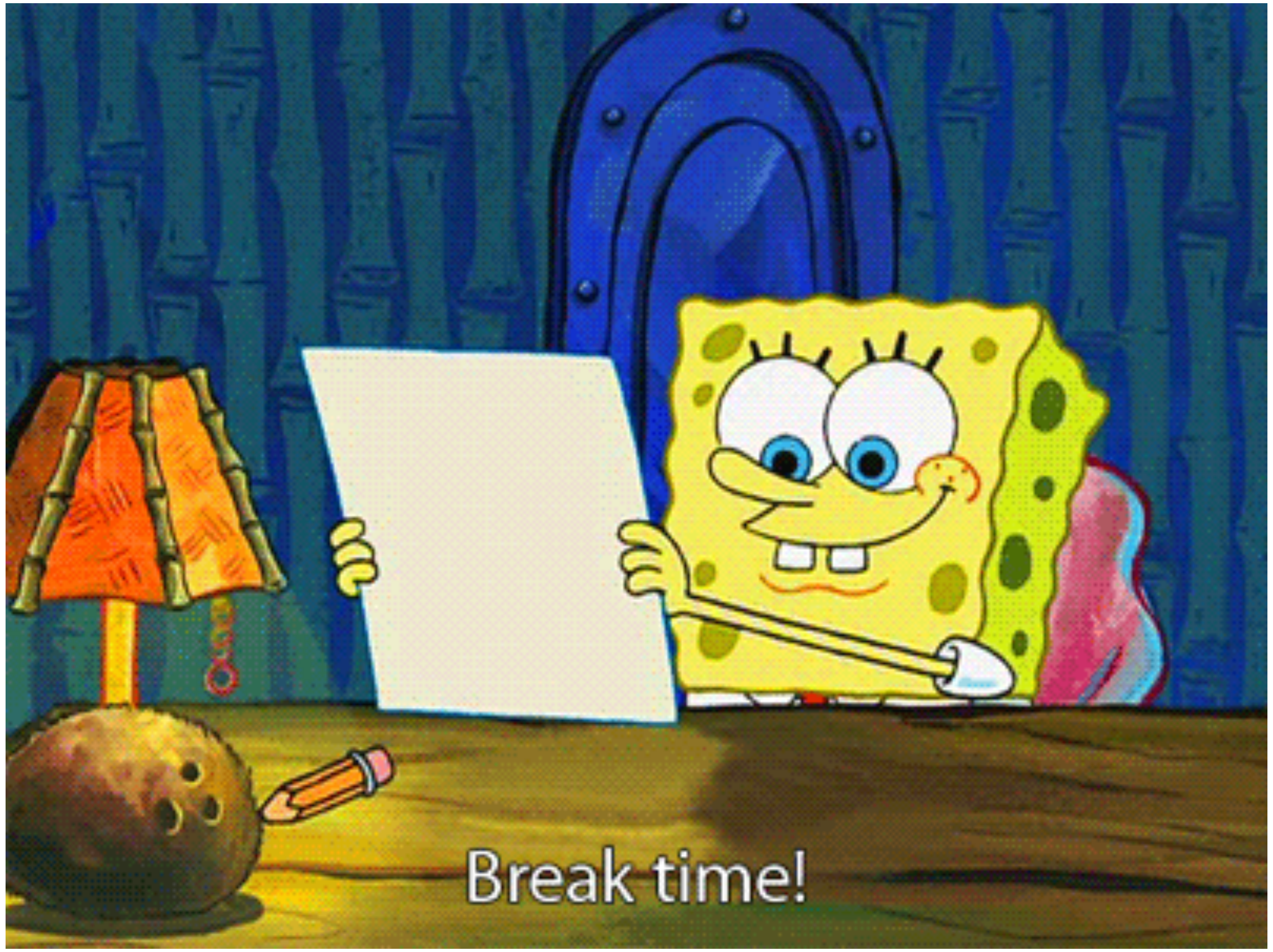
Summary (Pt. 2)

- A closure is a function with an extended scope that encompasses nonglobal variables referenced in the body of the function but not defined there.
  - A variable is `free` if the variable can be accessed outside the scope it was defined in.
  - A variable is `local` if it is defined within a scope
  - A `free` variable can become `local` if you attempt to modify the variable within the narrower scope, even if the variable was previously `free`
  - Uses the `nonlocal` keyword to access allow for modification of a `free` variable from within a narrower scope
- Decorators allow for reusability and promote a clean, concise coding style

Exercise 2 - Decorators and Closures

- In these exercises you will:
    - Implement a debugging decorator that prints the variables and results of a function
    - Implement a silly decorator that gives you the result of the previous operation
- Go to the Exercise Directory in the Google Drive and open the Practice Directory
- Open *02-Decorators-and-Closures.ipynb* and follow the instructions
- If you get stuck, raise your hand and someone will come by and help. You can also check the Solution directory for the answers
- You have **10 mins**

10 Minute Break



Part III: Comprehensions

Comprehensions

- Comprehensions provide a concise way to construct new sequences
  - Lists
  - Dictionaries
  - Sets
  - Generators
- Provides for better readability
- Better performance due to more optimized implementation

List Comprehensions

Creating a List of Even Numbers Using a Loop

Say we have the list `[1, 2, 3, 4, 5, 6]` and we wanted to create a new list containing all of the even numbers

You could do this with a loop:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = []

for x in nums:
    if x % 2 == 0:
        even_nums.append(x)

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Loop

Say we have the list `[1, 2, 3, 4, 5, 6]` and we wanted to create a new list containing all of the even numbers

You could do this with a loop:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = []

for x in nums:
    if x % 2 == 0:
        even_nums.append(x)

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Loop

Say we have the list `[1, 2, 3, 4, 5, 6]` and we wanted to create a new list containing all of the even numbers

You could do this with a loop:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = []

for x in nums:
    if x % 2 == 0:
        even_nums.append(x)

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Loop

Say we have the list `[1, 2, 3, 4, 5, 6]` and we wanted to create a new list containing all of the even numbers

You could do this with a loop:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = []

for x in nums:
    if x % 2 == 0:
        even_nums.append(x)

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Loop

Say we have the list `[1, 2, 3, 4, 5, 6]` and we wanted to create a new list containing all of the even numbers

You could do this with a loop:

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = []

for x in nums:
    if x % 2 == 0:
        even_nums.append(x)

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Comprehension

You can do the same as above using a List Comprehension

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = [x for x in nums if x % 2 == 0]

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Comprehension

You can do the same as above using a List Comprehension

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = [x for x in nums if x % 2 == 0]

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Comprehension

You can do the same as above using a List Comprehension

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = [x for x in nums if x % 2 == 0]

print(nums)
print(even_nums)
```

Creating a List of Even Numbers Using a Comprehension

You can do the same as above using a List Comprehension

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = [x for x in nums if x % 2 == 0]

print(nums)
print(even_nums)
```

Comprehension Layout

A Comprehension has three distinct parts:

- The variable result to store, with any operations **(Required)**
  - `x`
  - `x*2` would also be valid
- The iteration **(Required)**
  - `for x in nums`
- Conditional Logic *(Optional)*
  - `if x % 2 == 0`

Not *every* Comprehension requires all three parts. And some comprehensions may be comprised of multile of the sampe part.

For Example

```
nums = [1, 2, 3, 4, 5, 6]

# Multiply List by 2
x2 = [x*2 for x in nums]
print(x2)

# Get Even Nums
even = [x for x in nums if x %2 == 0]
print(even)

# Multiple every element in the list by every other element in the list
# in reverse
wat = [x * y for x in nums for y in nums[::-1]]
print(wat)
```

For Example

```
nums = [1, 2, 3, 4, 5, 6]

# Multiply List by 2
x2 = [x*2 for x in nums]
print(x2)

# Get Even Nums
even = [x for x in nums if x %2 == 0]
print(even)

# Multiple every element in the list by every other element in the list
# in reverse
wat = [x * y for x in nums for y in nums[::-1]]
print(wat)
```

For Example

```
nums = [1, 2, 3, 4, 5, 6]

# Multiply List by 2
x2 = [x*2 for x in nums]
print(x2)

# Get Even Nums
even = [x for x in nums if x %2 == 0]
print(even)

# Multiple every element in the list by every other element in the list
# in reverse
wat = [x * y for x in nums for y in nums[::-1]]
print(wat)
```

For Example

```
nums = [1, 2, 3, 4, 5, 6]

# Multiply List by 2
x2 = [x*2 for x in nums]
print(x2)

# Get Even Nums
even = [x for x in nums if x %2 == 0]
print(even)

# Multiple every element in the list by every other element in the list
# in reverse
wat = [x * y for x in nums for y in nums[::-1]]
print(wat)
```

For Example

```python
nums = [1, 2, 3, 4, 5, 6]

# Multiply List by 2
x2 = [x*2 for x in nums]
print(x2)

# Get Even Nums
even = [x for x in nums if x %2 == 0]
print(even)

# Multiple every element in the list by every other element in the list
# in reverse
wat = [x * y for x in nums for y in nums[::-1]]
print(wat)
```

Cartesian Products (Matrix Multiplication) with Loops

A common use case of list comprehensions is creating Cartesian Products, or the multiplication of two lists

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = []

for suit in suits:
    for rank in ranks:
        results.append(f"{rank} of {suit}")

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

A common use case of list comprehensions is creating Cartesian Products, or the multiplication of two lists

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = []

for suit in suits:
    for rank in ranks:
        results.append(f"{rank} of {suit}")

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

A common use case of list comprehensions is creating Cartesian Products, or the multiplication of two lists

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = []

for suit in suits:
    for rank in ranks:
        results.append(f"{rank} of {suit}")

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

A common use case of list comprehensions is creating Cartesian Products, or the multiplication of two lists

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = []

for suit in suits:
    for rank in ranks:
        results.append(f"{rank} of {suit}")

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

A common use case of list comprehensions is creating Cartesian Products, or the multiplication of two lists

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = []

for suit in suits:
    for rank in ranks:
        results.append(f"{rank} of {suit}")

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

A common use case of list comprehensions is creating Cartesian Products, or the multiplication of two lists

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = []

for suit in suits:
    for rank in ranks:
        results.append(f"{rank} of {suit}")

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

This can be simplified with a comprehension

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = [f"{rank} of {suit}" for suit in suits for rank in ranks]

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

This can be simplified with a comprehension

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = [f"{rank} of {suit}" for suit in suits for rank in ranks]

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

This can be simplified with a comprehension

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = [f"{rank} of {suit}" for suit in suits for rank in ranks]

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

This can be simplified with a comprehension

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = [f"{rank} of {suit}" for suit in suits for rank in ranks]

print(results)
```

Cartesian Products (Matrix Multiplication) with Loops

This can be simplified with a comprehension

```
suits = ["\u2663", "\u2665", "\u2666", "\u2660"]
ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

results = [f"{rank} of {suit}" for suit in suits for rank in ranks]

print(results)
```

Dictionary Comprehensions

Dictionary Comprehensions

Similar to how we do a List Comprehension, we can also do a dictionary comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

result = {}

for state, city in zip(states, cities):
    result[state] = city

print(result)
```

Dictionary Comprehensions

Similar to how we do a List Comprehension, we can also do a dictionary comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

result = {}

for state, city in zip(states, cities):
    result[state] = city

print(result)
```

Dictionary Comprehensions

Similar to how we do a List Comprehension, we can also do a dictionary comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

result = {}

for state, city in zip(states, cities):
    result[state] = city

print(result)
```

Dictionary Comprehensions

Similar to how we do a List Comprehension, we can also do a dictionary comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

result = {}

for state, city in zip(states, cities):
    result[state] = city

print(result)
```

Dictionary Comprehensions

Similar to how we do a List Comprehension, we can also do a dictionary comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

result = {}

for state, city in zip(states, cities):
    result[state] = city

print(result)
```

Now with a Comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

results = {city:state for (city, state) in zip(states, cities)}
print(results)
```

Now with a Comprehension

```
states = ["Texas", "New York", "Washington", "Ohio" ]
cities = ["Austin", "Albany", "Olympia", "Columbus"]

results = {city:state for (city, state) in zip(states, cities)}
print(results)
```

Another Example

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

results = {x:x**3 for x in nums if x % 2 != 0}

print(results)
```

Another Example

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

results = {x:x**3 for x in nums if x % 2 != 0}

print(results)
```

Set and Generator Comprehensions

Set and Generator Comprehensions

- As you've seen, the syntax for Comprehensions is the same regardless of sequence type.
- Sets and Generators are more niche, but still useful

Set Comprehensions

- Sets are a sequence in Python based on the mathematical Set.
- Sets are unordered, unchangeable, and unindexed
  - By unchangeable, you can remove or add, but not modify
- Duplicates are not allowed

Removing Duplicates from a List

```
original_list = [1, 2, 2, 3, 4, 4, 5]
list_without_duplicates = []

for item in original_list:
    if item not in list_without_duplicates:
        list_without_duplicates.append(item)

print(list_without_duplicates)
```

Removing Duplicates from a List

```python
original_list = [1, 2, 2, 3, 4, 4, 5]
list_without_duplicates = []

for item in original_list:
    if item not in list_without_duplicates:
        list_without_duplicates.append(item)

print(list_without_duplicates)
```

Removing Duplicates from a List

```python
original_list = [1, 2, 2, 3, 4, 4, 5]
list_without_duplicates = []

for item in original_list:
    if item not in list_without_duplicates:
        list_without_duplicates.append(item)

print(list_without_duplicates)
```

Removing Duplicates from a List with Set Comprehension

```
original_list = [1, 2, 2, 3, 4, 4, 5]

set_without_duplicates = {x for x in original_list}

print(set_without_duplicates)
```

Removing Duplicates from a List with Set Comprehension

```
original_list = [1, 2, 2, 3, 4, 4, 5]

set_without_duplicates = {x for x in original_list}

print(set_without_duplicates)
```

Removing Duplicates from a List with Set Comprehension

```python
original_list = [1, 2, 2, 3, 4, 4, 5]

set_without_duplicates = {x for x in original_list}

print(set_without_duplicates)
```

Removing Duplicates from a List with just the `Set()`

```python
original_list = [1, 2, 2, 3, 4, 4, 5]
list_without_duplicates = list(set(original_list))

print(list_without_duplicates)
```

Removing Duplicates from a List with just the Set()

```
original_list = [1, 2, 2, 3, 4, 4, 5]
list_without_duplicates = list(set(original_list))

print(list_without_duplicates)
```

Removing Duplicates from a List with just the Set()

```
original_list = [1, 2, 2, 3, 4, 4, 5]
list_without_duplicates = list(set(original_list))

print(list_without_duplicates)
```

Warning! Sets Do *NOT* Preserve Order

One thing to keep in mind is that sets **do not preserver order**. There is nothing guaranteeing that the order you see at one execution will be the same at the next.

Generator Comprehensions

- Generators don't allocate memory for the whole list
- The generator each value one by one
- Very useful if the comprehension you are trying to perform is on large sequences
- Represented using ( ) instead of [ ]

```
nums = (1, 2, 3, 4, 5, 6)
even_nums_gen = (x for x in nums if x % 2 == 0)

print(nums)
print(even_nums_gen)
for var in even_nums_gen:
    print(var, end = ' ')
```

Generator Comprehensions

- Generators don't allocate memory for the whole list
- The generator each value one by one
- Very useful if the comprehension you are trying to perform is on large sequences
- Represented using ( ) instead of [ ]

```
nums = (1, 2, 3, 4, 5, 6)
even_nums_gen = (x for x in nums if x % 2 == 0)

print(nums)
print(even_nums_gen)
for var in even_nums_gen:
    print(var, end = ' ')
```

Generator Comprehensions

- Generators don't allocate memory for the whole list
- The generator each value one by one
- Very useful if the comprehension you are trying to perform is on large sequences
- Represented using ( ) instead of [ ]

```
nums = (1, 2, 3, 4, 5, 6)
even_nums_gen = (x for x in nums if x % 2 == 0)

print(nums)
print(even_nums_gen)
for var in even_nums_gen:
    print(var, end = ' ')
```

Generator Comprehensions

- Generators don't allocate memory for the whole list
- The generator each value one by one
- Very useful if the comprehension you are trying to perform is on large sequences
- Represented using ( ) instead of [ ]

```
nums = (1, 2, 3, 4, 5, 6)
even_nums_gen = (x for x in nums if x % 2 == 0)

print(nums)
print(even_nums_gen)
for var in even_nums_gen:
    print(var, end = ' ')
```

Generator Comprehensions

- Generators don't allocate memory for the whole list
- The generator each value one by one
- Very useful if the comprehension you are trying to perform is on large sequences
- Represented using ( ) instead of [ ]

```
nums = (1, 2, 3, 4, 5, 6)
even_nums_gen = (x for x in nums if x % 2 == 0)

print(nums)
print(even_nums_gen)
for var in even_nums_gen:
    print(var, end = ' ')
```

Generator Comprehensions

- Generators don't allocate memory for the whole list
- The generator each value one by one
- Very useful if the comprehension you are trying to perform is on large sequences
- Represented using ( ) instead of [ ]

```
nums = (1, 2, 3, 4, 5, 6)
even_nums_gen = (x for x in nums if x % 2 == 0)

print(nums)
print(even_nums_gen)
for var in even_nums_gen:
    print(var, end = ' ')
```

Summary (Pt. 1)

- Comprehensions provide a concise way to construct new sequences
  - Lists
  - Dictionaries
  - Sets
  - Generators
- Provides for better readability
- Better performance due to more optimized implementation

Summary (Pt. 2)

- A Comprehension has three distinct parts:
  - The variable result to store, with any operations **(Required)**
    - `x`
    - `x*2` would also be valid
  - The iteration **(Required)**
    - `for x in nums`
  - Conditional Logic *(Optional)*
    - `if x % 2 == 0`

Summary (Pt. 3)

- Not *every* Comprehension requires all three parts. And some comprehensions may be comprised of multile of the sampe part.
- Set comprehensions are useful, but order is not preserved
- Generator comprehensions don't load the sequence in to memory, and instead generates it on demand, saving resources

Exercise 3 - Comprehensions

- In these exercises you will:
  - Implement a comprehension to return a list of people's initials given their names
  - Implement a comprehension to return a list of vowels in a string, with each vowel that is present only appearing in the result once
  - Implement a comprehension to return all possible class/race combinations from the lists of DND classes and races provided.
- Go to the Exercise Directory in the Google Drive and open the Practice Directory
- Open *03-Comprehensions.ipynb* and follow the instructions
- If you get stuck, raise your hand and someone will come by and help. You can also check the Solution directory for the answers
- You have **10 mins**

Part IV: Special Methods and Operator Overloading

Special Methods and Operator Overloading

Special or Magic or Dunder Methods

- Special, *Magic*, or *Dunder*, methods are special methods within Python associated with an object
- The term "dunder" comes from "double underscore", which is a characteristic of these methods
  - __init__
  - __str__
  - __len__
  - etc.
- There are many special methods in Python that are at the core of Python and how it supports its object-oriented features

Under the Hood

- Many operations within Python implicitly call magic methods to execute certain operations
- These methods are not intended to be directly called by you, but you can override them as we'll see later.

```
3 + 4
```

Under the hood this calls

```
(3).__add__(4)
```

A Few Magic Methods

Python Magic Methods

Class Instantiation	
`__init__(self, ...args)`	`ClassName()`
`__del__(self)`	`del instance`

Property Lookups	
`__getattr__(self, key)`	`instance.prop` (when `prop` not present)
`__getattribute__(self, key)`	`instance.prop` (regardless of `prop` present)
`__dir__(self)`	`dir(instance)`
`__setattr__(self, key, val)`	`instance.prop = newVal`
`__delattr__(self, key)`	`del instance.prop`
`__getitem__(self, key)`	`instance[prop]`
`__setitem__(self, key, val)`	`instance[prop] = newVal`
`__delitem__(self, key)`	`del instance[prop]`

List Iteration	
`__iter__(self)`	`[x for x in instance]`
`__contains__(self, item)`	`if x in instance`

Operator Overloads	
`__add__(self, other)`	`instance + other`
`__sub__(self, other)`	`instance - other`
`__mul__(self, other)`	`instance * other`
`__eq__(self, other)`	`instance == other`
`__ne__(self, other)`	`instance ≠ other`
`__lt__(self, other)`	`instance < other`
`__gt__(self, other)`	`instance > other`
`__le__(self, other)`	`instance ⩽ other`
`__ge__(self, other)`	`instance ⩾ other`

Type Casting	
`__bool__(self)`	`bool(instance)`
`__int__(self)`	`int(instance)`
`__str__(self)`	`str(instance)`

Controlling the Object Creation Process

- When you call a class constructer you create a new instance of that class
- When this happens Python invokes the `__new__()` method as the first step
  - This method is responsible for creating and returning a new empty object of this class
- This new object is then passed to `__init__()` to initialize the object with the appropriate values and properties
  - If you're familiar with OOP concepts, this is the Contstructor
  - Remember, all methods talk a first argument traditionally named `self`

`__init__()` Example

```python
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

mason = Person("Mason", "Egger")
print(mason)
```

__init__() Example

```
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

mason = Person("Mason", "Egger")
print(mason)
```

__init__() Example

```
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

mason = Person("Mason", "Egger")
print(mason)
```

__init__() Example

```python
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

mason = Person("Mason", "Egger")
print(mason)
```

Representing Objects as Strings

To represent the object as a human-readable string instead of the object reference, implement the `__str__()` method.

```
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __str__(self):
      return f"{self.first_name} {self.last_name}"

mason = Person("Mason", "Egger")
print(mason)
```

Representing Objects as Strings

To represent the object as a human-readable string instead of the object reference, implement the `__str__()` method.

```
class Person:

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __str__(self):
      return f"{self.first_name} {self.last_name}"

mason = Person("Mason", "Egger")
print(mason)
```

Making Your Objects Callable with **call**

You can implement the `__call__()` method to make your object callable after creation

```
class Factorial:
    def __init__(self):
        self._cache = {0: 1, 1: 1}

    def __call__(self, number):
        if number not in self._cache:
            self._cache[number] = number * self(number - 1)
        return self._cache[number]

factorial = Factorial()

print(factorial(4))
print(factorial(5))
print(factorial(6))
```

Considered by some to be an anti-pattern

Operator Overloading

Operator Overloading

Operator overloading is redefining the behavior of built-in operators for use with user-defined classes in Python.

This is a *very* powerful feature in programming languages and can easily lead to confusion and errors. Use caution when overloading operators.

A few things to remember:

- Cannot overload operators for the built-in types
- Cannot create new operators, only overload existing ones
- A few operators can't be overloaded
  - `is`, `and`, `or`, `not`
  - Although the bitwise operators can be

Overriding Mathematical Operators

- The operators +, −, ∗, /, etc. can be overridden for use with your custom object

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Point(new_x, new_y)

    def __str__(self):
        return f"Point ({self.x}, {self.y})"

x = Point(1, 2)
y = Point(3, 4)

x + y
```

Overriding Mathematical Operators

- The operators +, −, ∗, /, etc. can be overridden for use with your custom object

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Point(new_x, new_y)

    def __str__(self):
        return f"Point ({self.x}, {self.y})"

x = Point(1, 2)
y = Point(3, 4)

x + y
```

Overriding Mathematical Operators

- The operators +, −, ∗, /, etc. can be overridden for use with your custom object

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Point(new_x, new_y)

    def __str__(self):
        return f"Point ({self.x}, {self.y})"

x = Point(1, 2)
y = Point(3, 4)

x + y
```

Overriding Mathematical Operators

- The operators +, −, ∗, /, etc. can be overridden for use with your custom object

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Point(new_x, new_y)

    def __str__(self):
        return f"Point ({self.x}, {self.y})"

x = Point(1, 2)
y = Point(3, 4)

x + y
```

Overriding Mathematical Operators

- The operators +, −, ∗, /, etc. can be overridden for use with your custom object

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Point(new_x, new_y)

    def __str__(self):
        return f"Point ({self.x}, {self.y})"

x = Point(1, 2)
y = Point(3, 4)

x + y
```

Wait, why didn't my string get printed?

- `__str__()` produces a nice, human readable format when the object is being requested as a string, such as in a print statement
- `__repr__()` is more for developers. It is an unambiguous string representation and will be interpreted by the interpreter correctly. It should list enough information that you are able to recreate the object from it.
- When in doubt, implement both

Overloading Comparison Operators

- You can also overload comparison operators such as ==, !=, >, <, <=, etc.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

x = Point(1, 2)
y = Point(3, 4)
z = Point(1,2)

print(x == y)
print(x == z)
print(x == x)
print(z == x)
```

Overloading Comparison Operators

- You can also overload comparison operators such as ==, != , >, <, <=, etc.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

x = Point(1, 2)
y = Point(3, 4)
z = Point(1,2)

print(x == y)
print(x == z)
print(x == x)
print(z == x)
```

Overloading Comparison Operators

- You can also overload comparison operators such as ==, !=, >, <, <=, etc.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

x = Point(1, 2)
y = Point(3, 4)
z = Point(1,2)

print(x == y)
print(x == z)
print(x == x)
print(z == x)
```

Overloading Comparison Operators

- You can also overload comparison operators such as ==, != , >, <, <=, etc.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

x = Point(1, 2)
y = Point(3, 4)
z = Point(1,2)

print(x == y)
print(x == z)
print(x == x)
print(z == x)
```

Overloading Comparison Operators

- You can also overload comparison operators such as ==, != , >, <, <=, etc.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        return False

x = Point(1, 2)
y = Point(3, 4)
z = Point(1,2)

print(x == y)
print(x == z)
print(x == x)
print(z == x)
```

And so much more!

We only scratched the surface of special methods. There are 80+ special methods that allow you to control nearly every aspect of your objects. Visit the Python Documentation to learn about more.

Summary

- Special, *Magic*, or *Dunder*, methods are special methods within Python associated with an object
- The term "dunder" comes from "double underscore", which is a characteristic of these methods
- There are many magic methods in Python that are at the core of Python and how it supports its object-oriented features
- Many operations within Python implicitly call magic methods to execute certain operations
- These methods are not intended to be directly called by you, but you can override them to modify the functionality.

Exercise 4 - Special Methods

- In these exercises you will implement a Stack using only special methods
- Go to the Exercise Directory in the Google Drive and open the Practice Directory
- Open *04-Special-Methods-and-Operator-Overloading.ipynb* and follow the instructions
- If you get stuck, raise your hand and someone will come by and help. You can also check the Solution directory for the answers
- You have **15 mins**

10 Minute Break

Part V: Context Managers

Who's seen something like this before?

```
with open("file.txt", "w") as fh:
    text = fh.write("Hello")
```

Context Managers

- Also lovingly called the `with` block
- Flow control feature built into Python that's not often seen in other languages
- Sets up a temporary context and reliably tears it down
- Guarantee that some operation is performed both prior to and after a block of code, even in the case of an exception, return, or exit
- Allows for reusability, results in cleaner code, and is considered *Pythonic*

Possible Use Cases

- File Management
- Sessions
- Thread pools
- Locking
- Game environments (ppb)
- Mocking and Testing
- Logging
- And More!

Examples

Example 1

```
with open("file.txt", "r") as fh:
    text = fh.read()
```

Example 2

```
with ThreadPoolExecutor() as executor:
    for i in range(N):
        executor.submit(my_function, arg1, arg2)
```

Examples

Example 1

```
with open("file.txt", "r") as fh:
    text = fh.read()
```

Example 2

```
with ThreadPoolExecutor() as executor:
    for i in range(N):
        executor.submit(my_function, arg1, arg2)
```

Examples

Example 1

```
with open("file.txt", "r") as fh:
    text = fh.read()
```

Example 2

```
with ThreadPoolExecutor() as executor:
    for i in range(N):
        executor.submit(my_function, arg1, arg2)
```

Examples

Example 1

```
with open("file.txt", "r") as fh:
    text = fh.read()
```

Example 2

```
with ThreadPoolExecutor() as executor:
    for i in range(N):
        executor.submit(my_function, arg1, arg2)
```

More Examples

```python
async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        return await resp.text()

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

asyncio.run(main())
```

More Examples

```
async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        return await resp.text()

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

asyncio.run(main())
```

More Examples

```
async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        return await resp.text()

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

asyncio.run(main())
```

More Examples

```
async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        return await resp.text()

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

asyncio.run(main())
```

More Examples

```
async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        return await resp.text()

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

asyncio.run(main())
```

More Examples

```
async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        return await resp.text()

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

asyncio.run(main())
```

Create Your Own Context Manager

- Creating a class and defining the `__enter__` and `__exit__` special methods
- Creating a function and using the `contextlib` library

Implementing a Context Manager as a Class

Context Manager as a Class `__enter__()`

- The `__enter__` magic method is invoked at the start of execution on the context manager object
- All code within the the `__enter__` method is executed prior to the code within the block
- Can only have self as a parameter

```
class MyContextManager:

    def __enter__(self):
        print("Hello")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

with MyContextManager():
    print("hi")
```

Context Manager as a Class `__enter__()`

- The `__enter__` magic method is invoked at the start of execution on the context manager object
- All code within the the `__enter__` method is executed prior to the code within the block
- Can only have self as a parameter

```
class MyContextManager:

    def __enter__(self):
        print("Hello")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

with MyContextManager():
    print("hi")
```

`__enter__()` Return Value

- The `__enter__()` method may return an object
- The value will be returned when invoking the Context Manager

```
class MyContextManager:

    def __enter__(self):
        print("Hello")
        return("Hola")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

with MyContextManager() as cm:
    print("hi")
print(cm)
```

Passing Parameters to your Context Manager

- Context Managers are classes, and creating an instance of the Context Manager will invoke `__init__()`
- Pass any parameters you'd like to include in your Context Manager into `__init__()`

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

with MyContextManager("Mason") as cm:
    print("hi")
print(cm)
```

Passing Parameters to your Context Manager

- Context Managers are classes, and creating an instance of the Context Manager will invoke `__init__()`
- Pass any parameters you'd like to include in your Context Manager into `__init__()`

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

with MyContextManager("Mason") as cm:
    print("hi")
print(cm)
```

Passing Parameters to your Context Manager

- Context Managers are classes, and creating an instance of the Context Manager will invoke `__init__()`
- Pass any parameters you'd like to include in your Context Manager into `__init__()`

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

with MyContextManager("Mason") as cm:
    print("hi")
print(cm)
```

__exit__()

- The `__exit__()` special method is invoked after the execution of the body of the Context Manager

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type, exc_value, traceback):
        print("Finished")

with MyContextManager("Mason") as cm:
    print("hi")
print(cm)
```

`__exit__()`

- The `__exit__()` special method is invoked after the execution of the body of the Context Manager

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type, exc_value, traceback):
        print("Finished")

with MyContextManager("Mason") as cm:
    print("hi")
print(cm)
```

`__exit__()` Exceptions

- `__exit__()` returns a Boolean flag indicating if an exception that occurred should be suppressed
  - If `True`, the exception will be suppressed.
  - Otherwise the exception will continue propagating up.

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type,
                 exc_value, traceback):
        print("Finished")
        return True

with MyContextManager("Mason") as cm:
    print("hi")
    raise Exception
print(cm)
```

`__exit__()` Exceptions

- `__exit__()` returns a Boolean flag indicating if an exception that occurred should be suppressed
  - If `True`, the exception will be suppressed.
  - Otherwise the exception will continue propagating up.

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type,
                 exc_value, traceback):
        print("Finished")
        return True

with MyContextManager("Mason") as cm:
    print("hi")
    raise Exception
print(cm)
```

`__exit__()` Parameters

- `__exit()__` takes three arguments
  - `exc_type` - The exception class
  - `exc_val` - The exception instance
  - `traceback` - A traceback object

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type,
                 exc_value, traceback):
        safe_exception = False
        if exc_type is ZeroDivisionError:
            print(f"Exception: {exc_value}")
            safe_exception = True
        print("Finished")
        return safe_exception

with MyContextManager("Mason") as cm:
    print("hi")
    1/0
print(cm)
```

`__exit__()` Parameters

- `__exit()__` takes three arguments
  - `exc_type` - The exception class
  - `exc_val` - The exception instance
  - `traceback` - A traceback object

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type,
                 exc_value, traceback):
        safe_exception = False
        if exc_type is ZeroDivisionError:
            print(f"Exception: {exc_value}")
            safe_exception = True
        print("Finished")
        return safe_exception

with MyContextManager("Mason") as cm:
    print("hi")
    1/0
print(cm)
```

__exit__() Parameters

- __exit()__ takes three arguments
  - exc_type - The exception class
  - exc_val - The exception instance
  - traceback - A traceback object

```
class MyContextManager:

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"Hello {self.name}")
        return("Hola")

    def __exit__(self, exc_type,
                 exc_value, traceback):
        safe_exception = False
        if exc_type is ZeroDivisionError:
            print(f"Exception: {exc_value}")
            safe_exception = True
        print("Finished")
        return safe_exception

with MyContextManager("Mason") as cm:
    print("hi")
    1/0
print(cm)
```

Implementing a Context Manager as a Function Using `contextlib`

Context Manager as a Function with `contextlib`

- Anther way to implement a Context Manager is through the use of functions, generators, and the `contextlib` library
- Use the `@contextlib.contextmanager` decorator to designate a function as a context manager
- Use the `yield` builtin to separate the *enter* and *exit* sections

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Hola"
    print("Finished")

with my_context_manager("Mason") as cm:
    print("hi")
print(cm)
```

Context Manager as a Function with `contextlib`

- Anther way to implement a Context Manager is through the use of functions, generators, and the `contextlib` library
- Use the `@contextlib.contextmanager` decorator to designate a function as a context manager
- Use the `yield` builtin to separate the *enter* and *exit* sections

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Hola"
    print("Finished")

with my_context_manager("Mason") as cm:
    print("hi")
print(cm)
```

Context Manager as a Function with `contextlib`

- Anther way to implement a Context Manager is through the use of functions, generators, and the `contextlib` library
- Use the `@contextlib.contextmanager` decorator to designate a function as a context manager
- Use the `yield` builtin to separate the *enter* and *exit* sections

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Hola"
    print("Finished")

with my_context_manager("Mason") as cm:
    print("hi")
print(cm)
```

Context Manager as a Function with `contextlib`

- Anther way to implement a Context Manager is through the use of functions, generators, and the `contextlib` library
- Use the `@contextlib.contextmanager` decorator to designate a function as a context manager
- Use the `yield` builtin to separate the *enter* and *exit* sections

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Hola"
    print("Finished")

with my_context_manager("Mason") as cm:
    print("hi")
print(cm)
```

Context Manager as a Function with `contextlib`

- Anther way to implement a Context Manager is through the use of functions, generators, and the `contextlib` library
- Use the `@contextlib.contextmanager` decorator to designate a function as a context manager
- Use the `yield` builtin to separate the *enter* and *exit* sections

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Hola"
    print("Finished")

with my_context_manager("Mason") as cm:
    print("hi")
print(cm)
```

Context Manager as a Function with `contextlib`

- Anther way to implement a Context Manager is through the use of functions, generators, and the `contextlib` library
- Use the `@contextlib.contextmanager` decorator to designate a function as a context manager
- Use the `yield` builtin to separate the *enter* and *exit* sections

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Hola"
    print("Finished")

with my_context_manager("Mason") as cm:
    print("hi")
print(cm)
```

Comparisson between Class and contextlib

```python
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
    print(f"Hello {name}")
    yield "Butters"
    print("Finished")
```

```python
class MyContextManager:
    def __init__(self, name):
        self.name = name
    def __enter__(self):
        print(f"Hello {self.name}")
        return "Butters"
    def __exit__(self, exc_type, exc_value, traceback):
        print("Finished")
```

Context Manager as a Function with `contextlib` Exceptions

- Handle exceptions with `try`/`except`/`finally`

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
  print(f"Hello {name}")
  try:
    yield "Hola"
  except Exception as e:
    print(f"Exception occurred: {e}")
  finally:
    print("Finished")

with my_context_manager("Mason") as cm:
  print("hi")
  raise Exception("Oops")

print(cm)
```

Context Manager as a Function with `contextlib` Exceptions

- Handle exceptions with `try`/`except`/`finally`

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
  print(f"Hello {name}")
  try:
    yield "Hola"
  except Exception as e:
    print(f"Exception occurred: {e}")
  finally:
    print("Finished")

with my_context_manager("Mason") as cm:
  print("hi")
  raise Exception("Oops")

print(cm)
```

Context Manager as a Function with `contextlib` Exceptions

- Handle exceptions with `try`/`except`/`finally`

```
import contextlib

@contextlib.contextmanager
def my_context_manager(name):
  print(f"Hello {name}")
  try:
    yield "Hola"
  except Exception as e:
    print(f"Exception occurred: {e}")
  finally:
    print("Finished")

with my_context_manager("Mason") as cm:
  print("hi")
  raise Exception("Oops")

print(cm)
```

Summary

- Context managers are a flow control mechanism that sets up a temporary context and reliably tears it down.
- Two ways of implementing:
  - As a Class, using the **enter** and **exit** magic methods
    - Passing in a variable is done via **init**
    - Returning True from **exit** will supress Exceptions raised in the invocation
  - As a decorated function using contextlib
    - Entrance and exit code separate by a yield statement that provides the value assigned to the variable in the as clause
    - Exceptions handled with a try/except/finally
- If you make changes during the system within the scope of the context manager, be sure to set the back

Exercise

- In these exercises you will custom context manager that reads in a file and prints it in reverse
- Go to the Exercise Directory in the Google Drive and open the Practice Directory
- Open *05-Context-Managers.ipynb* and follow the instructions
- If you get stuck, raise your hand and someone will come by and help. You can also check the Solution directory for the answers
- You have **15 mins**

Workshop Summary

Summary (Pt. 1)

- Functions are considered First-Class in Python
- First-Class means that the function is treated like an object
- Just like other objects, functions can be:
    - Created at runtime
    - Assigned to a variable or element in a data structure
    - Passed as an argument to a function
    - Returned as the result of a function

Summary (Pt. 2)

- Functions that take other functions as parameters, or return a function as a result is known as a *Higher-Order Function*
- Anonymous functions are implemented using the `lambda` keyword, and are good for creating concise, one off functions.

Summary (Pt. 3)

- Decorators allow us to "mark" functions to enhance their behavior
- Decorators are syntactic sugar for Higher-Order Functions
- Decorators return an entirely new function that may or may not call the original function
- Decorators are first run at *import time*

Summary (Pt. 4)

- A closure is a function with an extended scope that encompasses nonglobal variables referenced in the body of the function but not defined there.
  - A variable is `free` if the variable can be accessed outside the scope it was defined in.
  - A variable is `local` if it is defined within a scope
  - A `free` variable can become `local` if you attempt to modify the variable within the narrower scope, even if the variable was previously `free`
  - Uses the `nonlocal` keyword to access allow for modification of a `free` variable from within a narrower scope
- Decorators allow for reusability and promote a clean, concise coding style

Summary (Pt. 5)

- Comprehensions provide a concise way to construct new sequences
  - Lists
  - Dictionaries
  - Sets
  - Generators
- Provides for better readability
- Better performance due to more optimized implementation

Summary (Pt. 6)

- A Comprehension has three distinct parts:
  - The variable result to store, with any operations **(Required)**
    - `x`
    - `x*2` would also be valid
  - The iteration **(Required)**
    - `for x in nums`
  - Conditional Logic *(Optional)*
    - `if x % 2 == 0`

Summary (Pt. 7)

- Not *every* Comprehension requires all three parts. And some comprehensions may be comprised of multile of the sampe part.
- Set comprehensions are useful, but order is not preserved
- Generator comprehensions don't load the sequence in to memory, and instead generates it on demand, saving resources

Summary (Pt. 8)

- Special, *Magic*, or *Dunder*, methods are special methods within Python associated with an object
- The term "dunder" comes from "double underscore", which is a characteristic of these methods
- There are many magic methods in Python that are at the core of Python and how it supports its object-oriented features
- Many operations within Python implicitly call magic methods to execute certain operations
- These methods are not intended to be directly called by you, but you can override them to modify the functionality.

Summary (Pt. 9)

- Context managers are a flow control mechanism that sets up a temporary context and reliably tears it down.
- Two ways of implementing:
  - As a Class, using the **enter** and **exit** magic methods
    - Passing in a variable is done via **init**
    - Returning True from **exit** will supress Exceptions raised in the invocation
  - As a decorated function using contextlib
    - Entrance and exit code separate by a yield statement that provides the value assigned to the variable in the as clause
    - Exceptions handled with a try/except/finally
- If you make changes during the system within the scope of the context manager, be sure to set the back

Thank You

- Thank you for being part of Tutorials at PyTexas
  - This is the first time we've done Tutorials since 2017
- You can find me on the socials mason.dev/links
- If your interested in learning Durable Execution and Temporal, check out learn.temporal.io
  - Maybe you just want to learn more from me