ERRORS AND EXCEPTIONS

The Zen Of Python Principles
10 and 11...

Error Handling

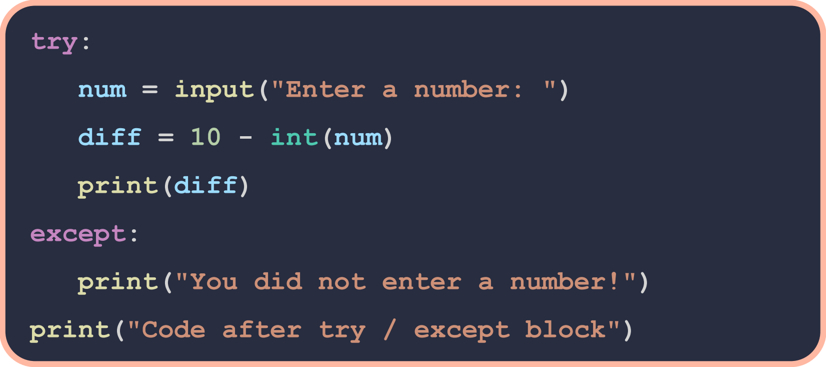
➔ When an error in the code occurs, Python execution ceases and a log of the error is printed to the console

```python
num = input("Enter a number: ")
diff = 10 - int(num)
print(diff)
```

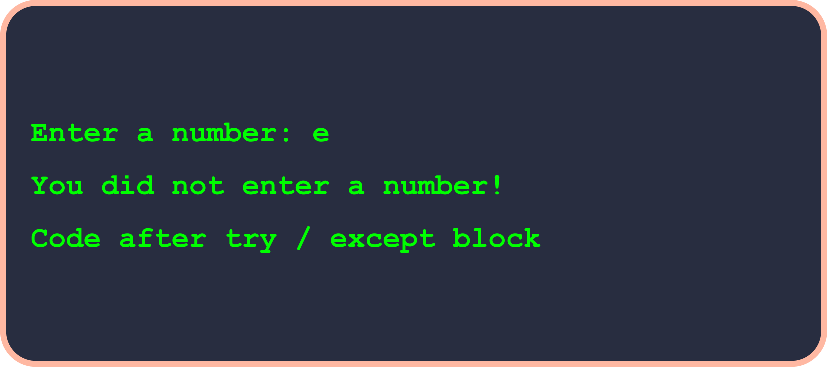
```
Enter a number: e
Traceback (most recent call last):
  File "<filepath>.py", line 2, in <module>
    diff = 10 - int(num)
ValueError: invalid literal for int() with
base 10: 'e'
```

Try / Except

➔ If an error occurs within a **try block**, the entire block is skipped and the code within the **except block** is executed



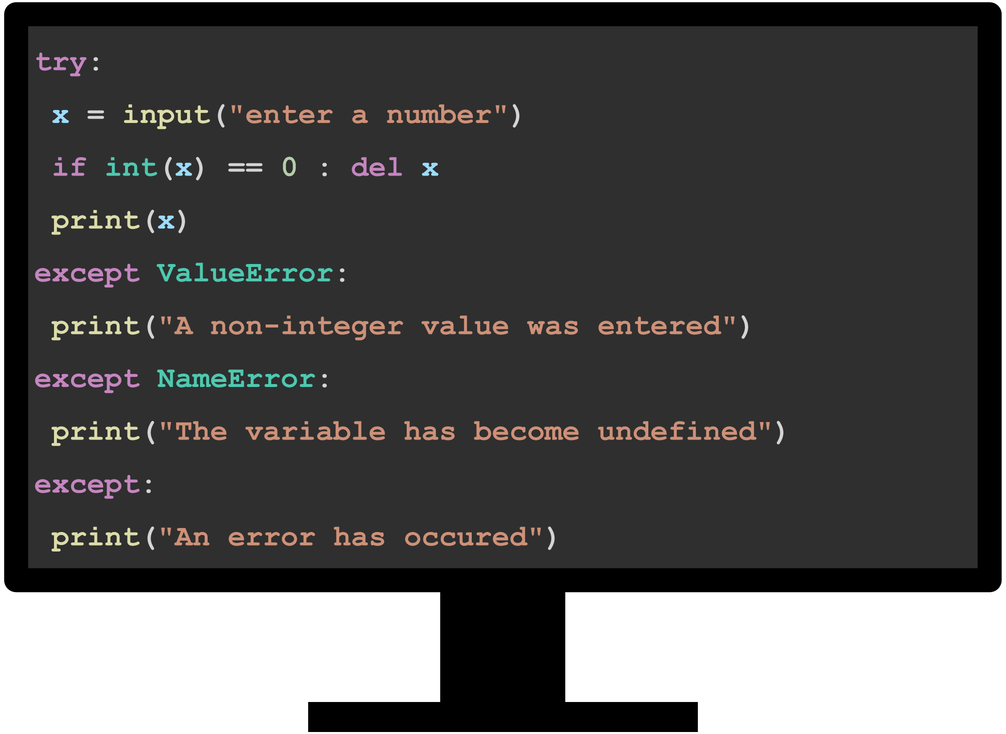
```
try:
   num = input("Enter a number: ")
   diff = 10 - int(num)
   print(diff)
except:
   print("You did not enter a number!")
print("Code after try / except block")
```



```
Enter a number: e
You did not enter a number!
Code after try / except block
```

Multi-Except

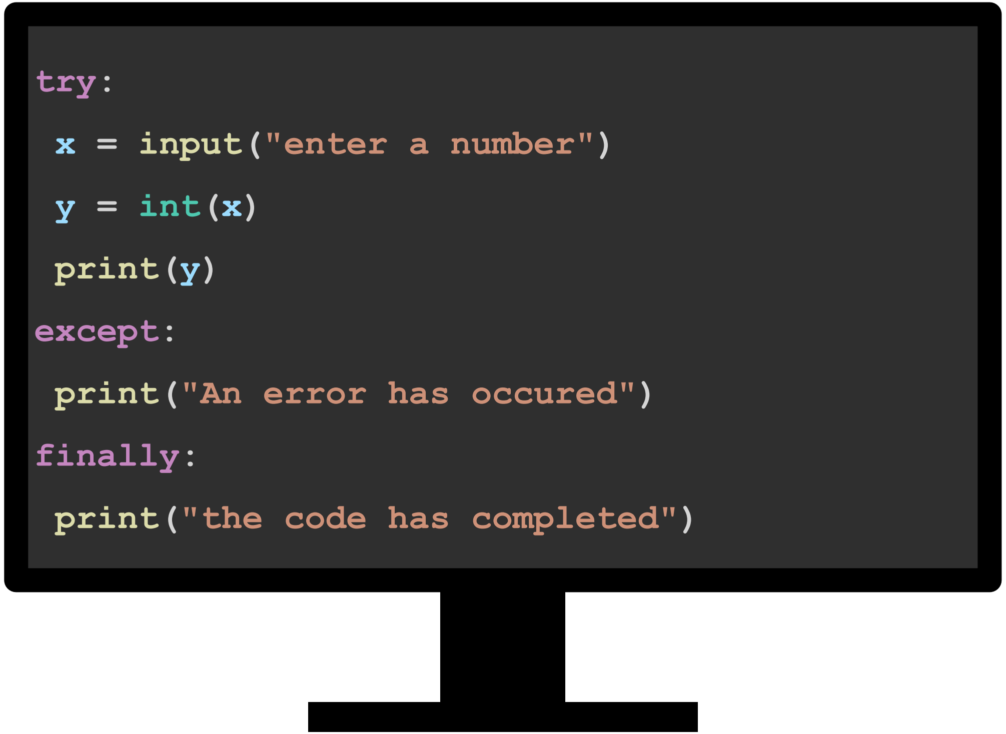
- ➔ Specific **errors** can be caught in their own except blocks
  - ◆ Each **error** can be handled differently
- ➔ An **except block** without a named error will catch any **error**
  - ◆ The general catch must be placed after any named **error**



```
try:
 x = input("enter a number")
 if int(x) == 0 : del x
 print(x)
except ValueError:
 print("A non-integer value was entered")
except NameError:
 print("The variable has become undefined")
except:
 print("An error has occured")
```

Finally

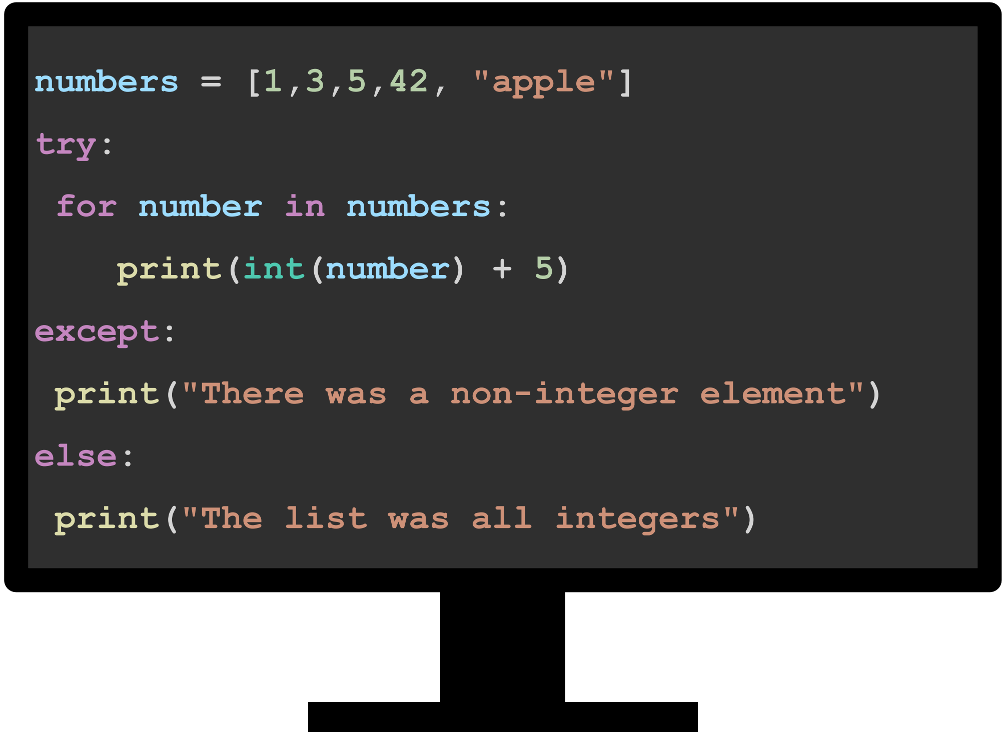
- A **finally** block will execute regardless of the status of the try/except blocks
- A **finally** block is a good place to perform any cleanup
  - Close connections
  - End timers
  - Cancel subscriptions



```python
try:
 x = input("enter a number")
 y = int(x)
 print(y)
except:
 print("An error has occured")
finally:
 print("the code has completed")
```

Else

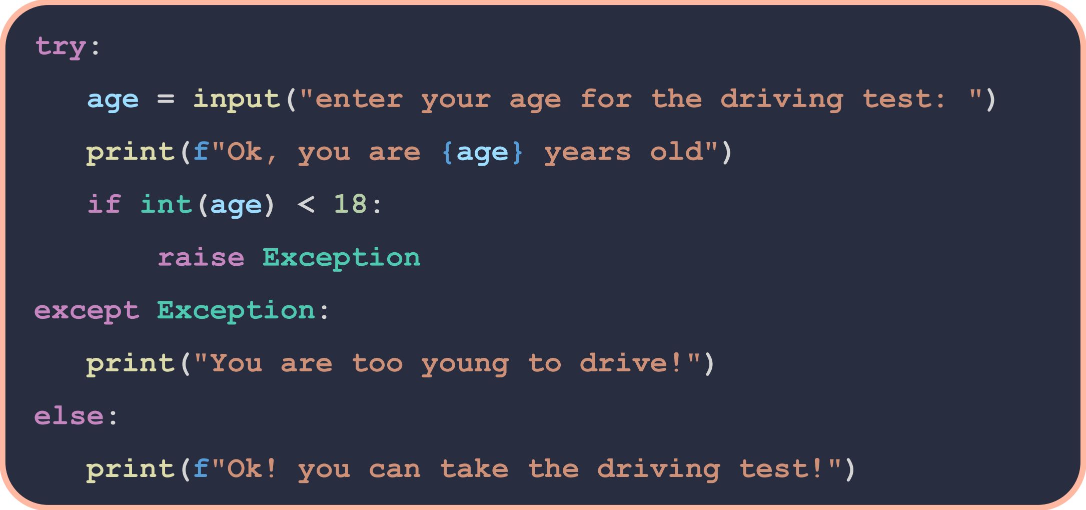
- An **else block** will only execute if the **try block** executed without **error**
- An **else block** is a good place to perform any actions that depend on the successful completion of the **try block**



```
numbers = [1,3,5,42, "apple"]
try:
 for number in numbers:
    print(int(number) + 5)
except:
 print("There was a non-integer element")
else:
 print("The list was all integers")
```

Raise

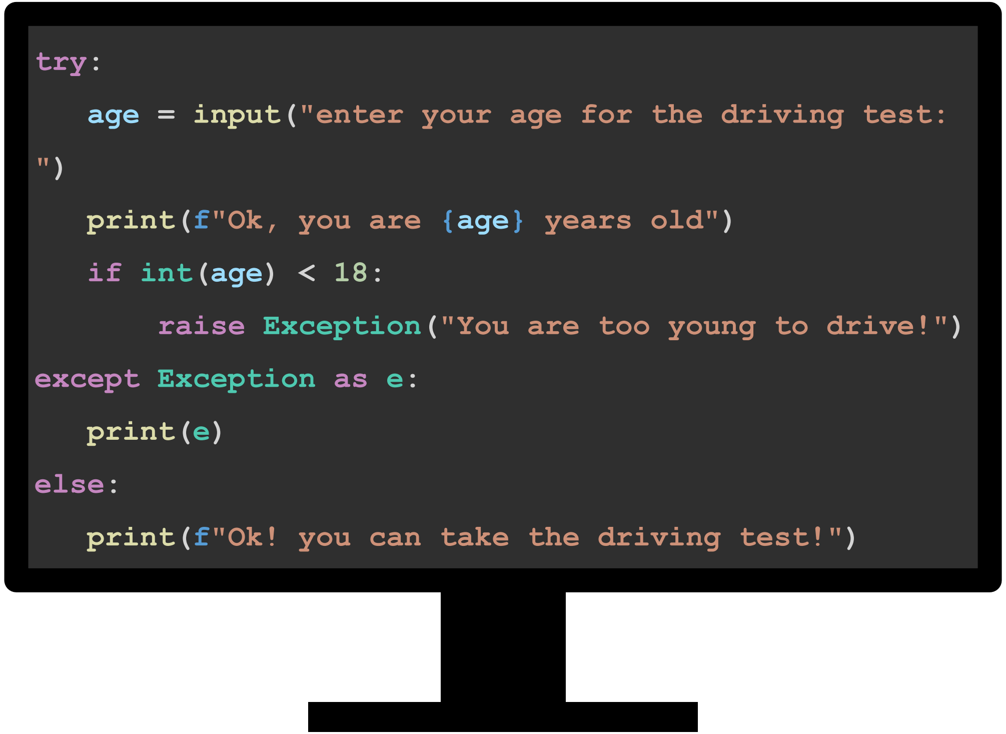
- ➔ **Errors** or **Exceptions** can be manually thrown with **raise** keyword



```python
try:
    age = input("enter your age for the driving test: ")
    print(f"Ok, you are {age} years old")
    if int(age) < 18:
        raise Exception
except Exception:
    print("You are too young to drive!")
else:
    print(f"Ok! you can take the driving test!")
```

Exception Message

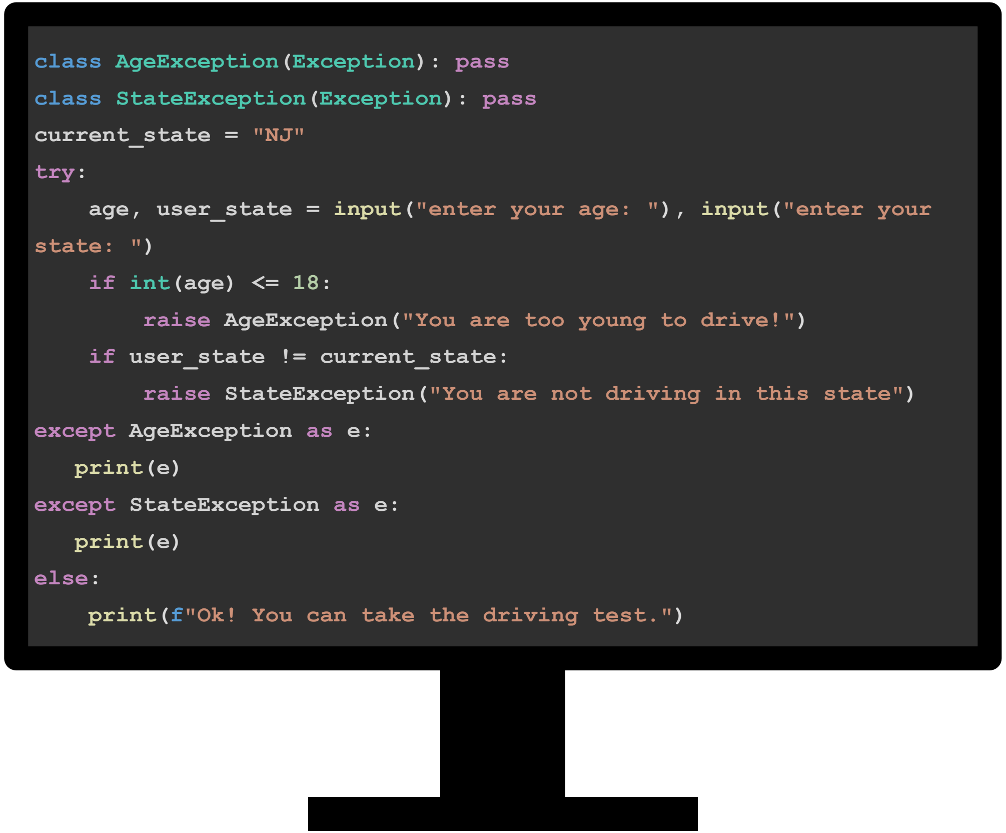
- ➔ A raised **Exception** can be passed a string argument as a message
  - ◆ This message will be printed as the error log
- ➔ A caught **Exception** can be aliased using **as**
  - ◆ Parsing this object to a string will return the message



```
try:
   age = input("enter your age for the driving test: 
")
   print(f"Ok, you are {age} years old")
   if int(age) < 18:
       raise Exception("You are too young to drive!")
except Exception as e:
   print(e)
else:
   print(f"Ok! you can take the driving test!")
```

Custom Exception

- A custom **Exception** class ban be created by extending Exception
- A custom Exceptions has a different **name**
  - This means it can be **caught** separately
- A custom **Exception** can have custom data or behaviors
- An empty custom Exception has the default behaviors but can still be **caught** separately



```
class AgeException(Exception): pass
class StateException(Exception): pass
current_state = "NJ"
try:
    age, user_state = input("enter your age: "), input("enter your
state: ")
    if int(age) <= 18:
        raise AgeException("You are too young to drive!")
    if user_state != current_state:
        raise StateException("You are not driving in this state")
except AgeException as e:
   print(e)
except StateException as e:
   print(e)
else:
    print(f"Ok! You can take the driving test.")
```

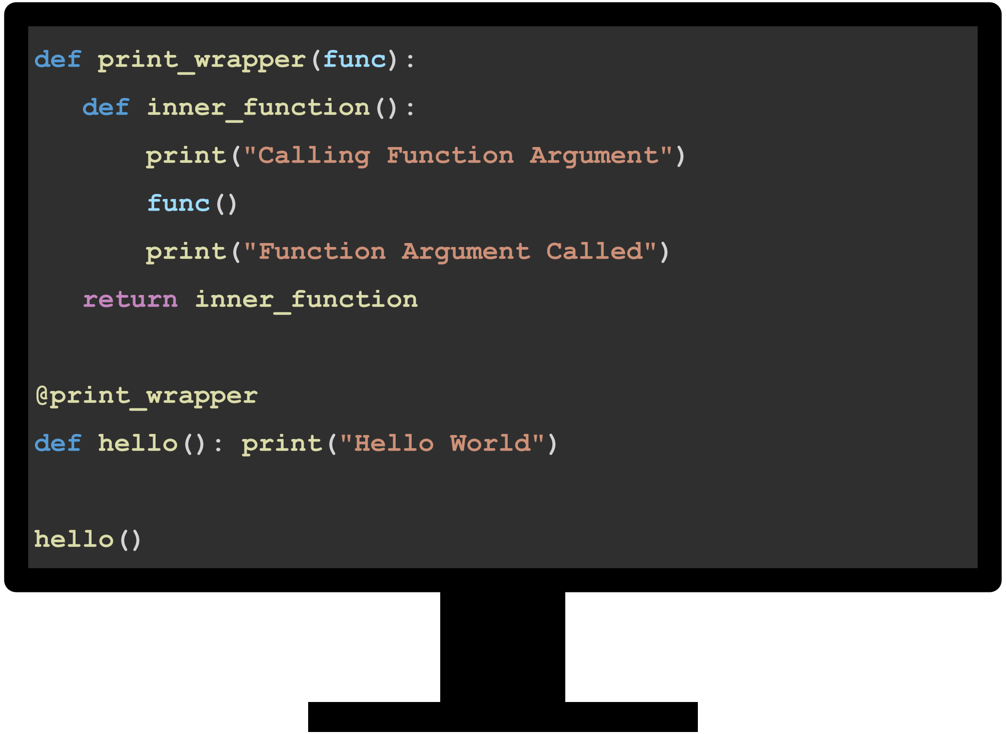
Student Exercise

- Take your Employee program and expand on it
- Create a list to hold your employees
  - Each employee should be a dictionary
- Prompt the user to say how many employees they will add
  - Use error handling to repeat the prompt until an integer is entered
  - Optional: add a max number of employees
- Loop for each employee and record their information
  - Use error handling to repeat the prompt until an integer is entered for age
- Print each employee's information in a formatted string

Function Decorators

Decorators

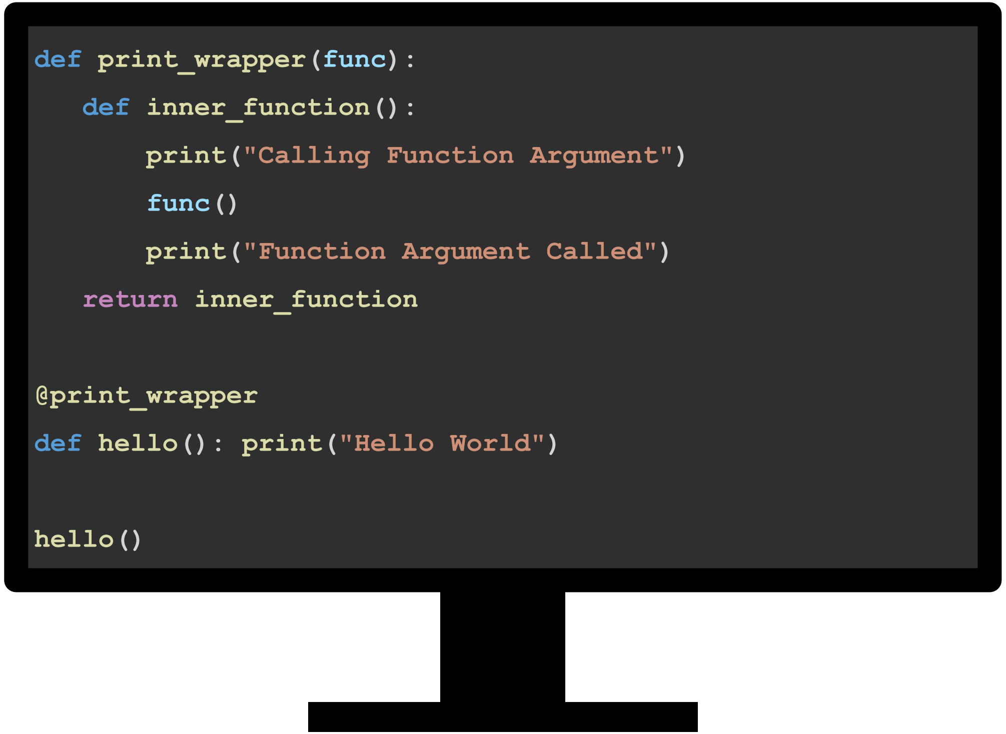
- ➔ Decorators are **Syntactic Sugar** passing a function to another function
- ➔ Decorators are declared with the **@** symbol
- ➔ **Functions**, **Methods**, or **Classes** can be decorated
- ➔ Some decorators can be passed **arguments**



```
def print_wrapper(func):
   def inner_function():
       print("Calling Function Argument")
       func()
       print("Function Argument Called")
   return inner_function

@print_wrapper
def hello(): print("Hello World")

hello()
```

Decorators

- ➔ A concise definition of a decorator is
  - ◆ **A decorator modifies a function's behavior without changing its name**
- ➔ Many decorators that are used are predefined in the **Python environment** or in **imported libraries**

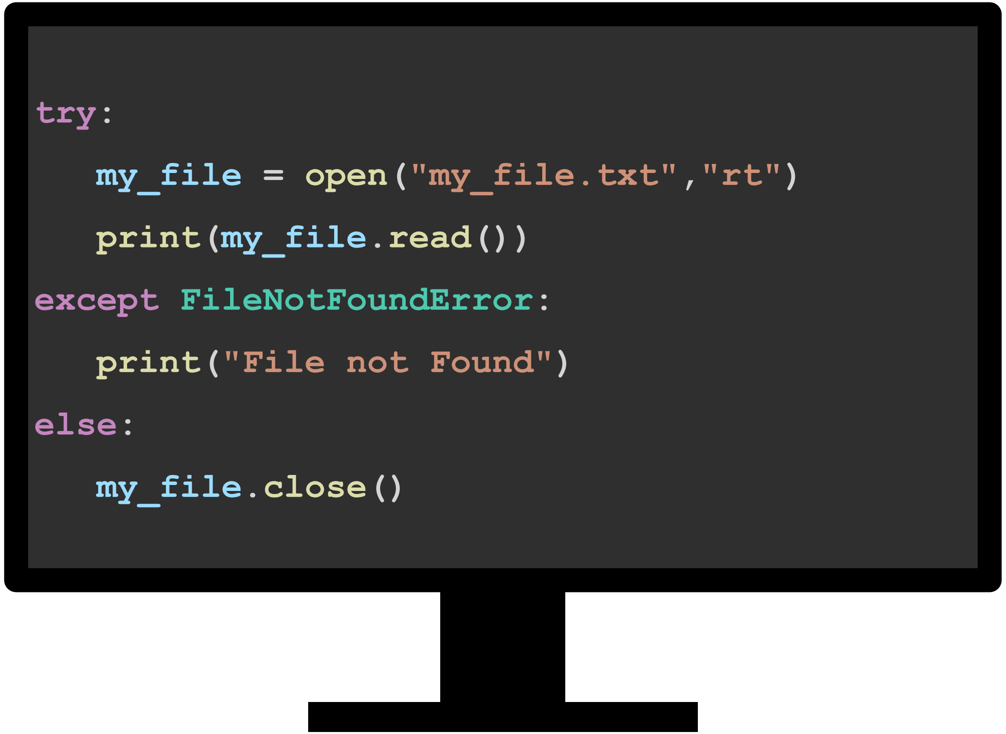


```
def print_wrapper(func):
   def inner_function():
       print("Calling Function Argument")
       func()
       print("Function Argument Called")
   return inner_function

@print_wrapper
def hello(): print("Hello World")

hello()
```

File Reading

open Function

- ➔ Globally available function
- ➔ The **open()** function will access a file
- ➔ Passed a file path and an open mode
- ➔ Will raise a **FileNotFoundError** if the file does not found
- ➔ The file should be closed after use



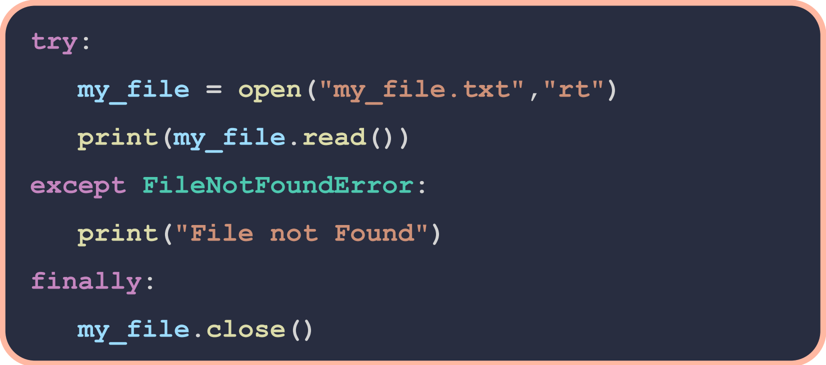
```
try:
    my_file = open("my_file.txt","rt")
    print(my_file.read())
except FileNotFoundError:
    print("File not Found")
else:
    my_file.close()
```

open Modes

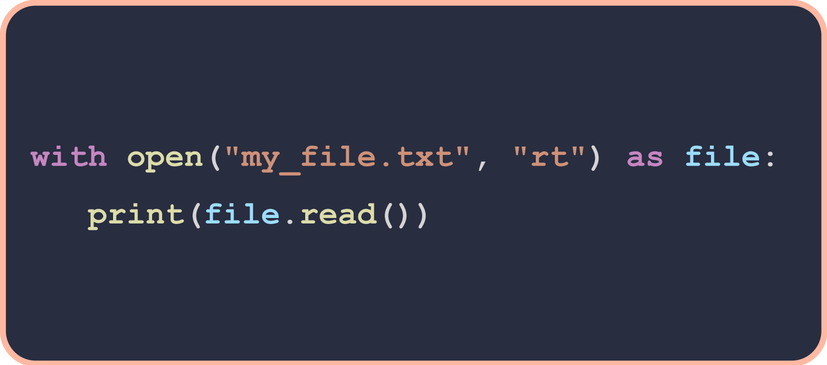
- **r***
  - Read a file (default)
- **a***
  - Append to a file
- **w***
  - Override a file
- **x***
  - Create a file
- ***t**
  - Read/Write text (default)
- ***b**
  - Read/Write binary data

with open

- ➔ The **with** keyword assists with error handling
- ➔ The file is automatically closed once the end of the **with block** is reached



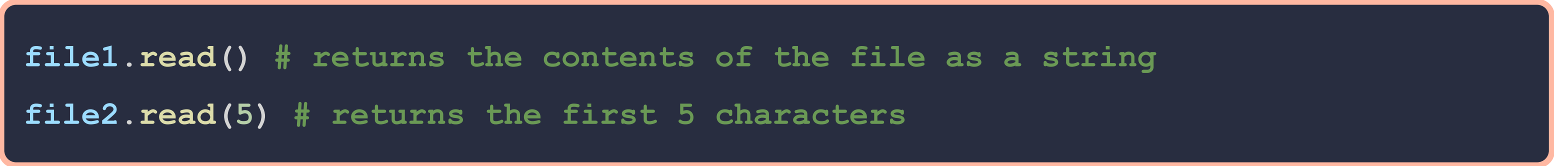
```python
try:
    my_file = open("my_file.txt","rt")
    print(my_file.read())
except FileNotFoundError:
    print("File not Found")
finally:
    my_file.close()
```



```python
with open("my_file.txt", "rt") as file:
    print(file.read())
```

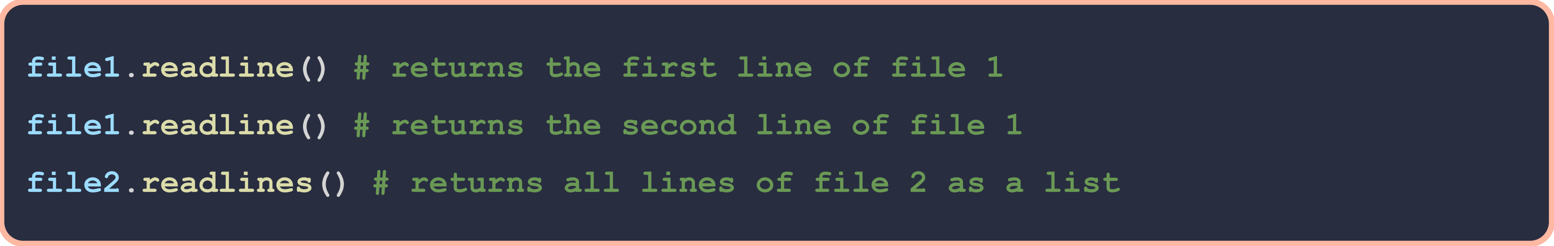
File Reading

- The .read() method of a file object will read the contents of a file
  - Passing a number argument to .read() will read that number of characters



```
file1.read() # returns the contents of the file as a string
file2.read(5) # returns the first 5 characters
```

- .readline() will read the next line in a file
- .readlines() will return the lines of the file as a list



```
file1.readline() # returns the first line of file 1
file1.readline() # returns the second line of file 1
file2.readlines() # returns all lines of file 2 as a list
```

File Creating and Writing

- ➔ The “x” mode of the open() function will create a new file at the given file path
    - ◆ It will throw a **FileExistsError** if the file already exists

```
new_file = open("newfile.txt","x")
```

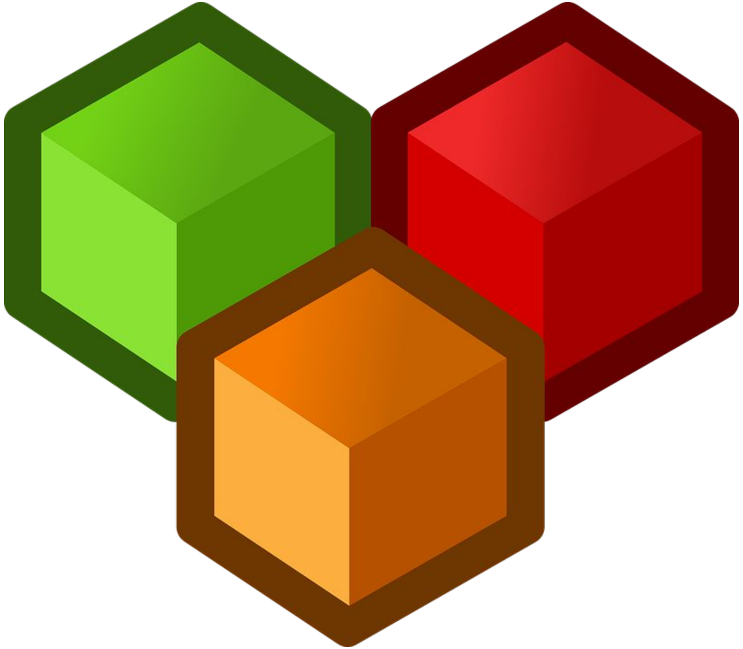
- ➔ The “w” mode of the open() function will overwrite the file at the given file path
- ➔ The “a” mode of the open() function will append to the file at the given file path
- ➔ .write(<str>) will write the given string to the file
- ➔ .writelines(<iterable>) will write each element of an iterable to

```
new_file = open("newfile.txt","w")
new_file.write("Hello")
new_file.writelines(["1","2","3"])
```

Student Exercise

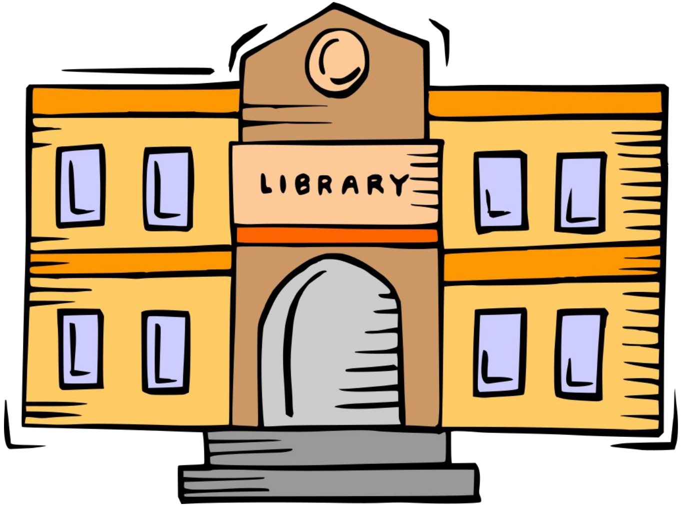
- Further expand your Employee Exercise
- Instead of printing the data to the screen, append the data to a file
  - Create the file in your code if it doesn't exist
  - Append the data to it if the file exists

Python Modules



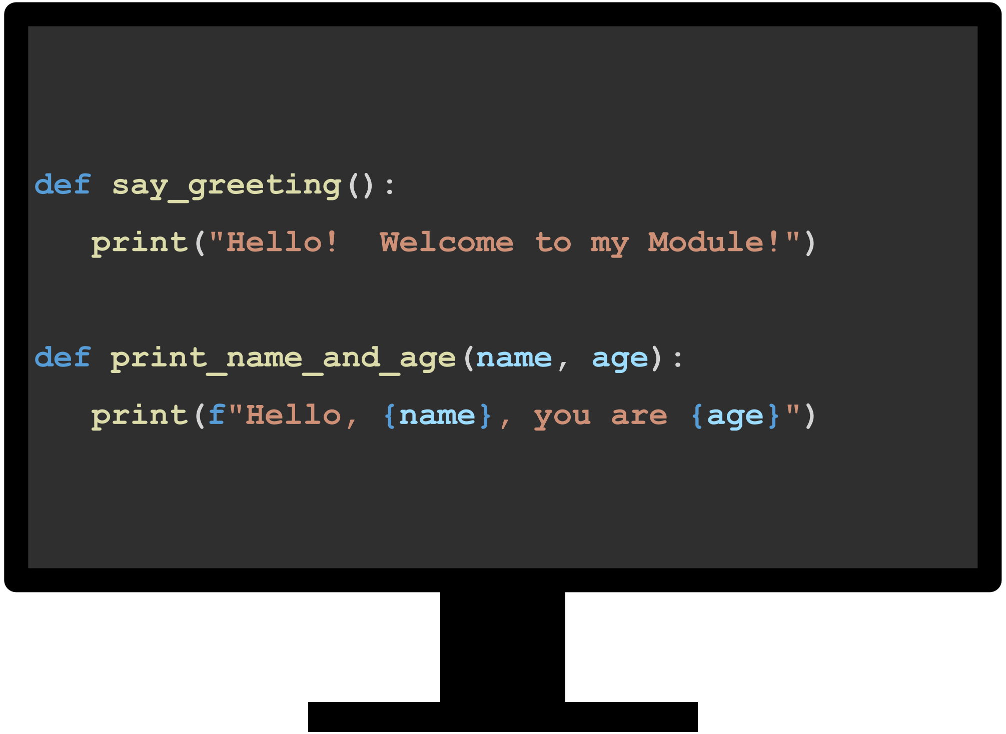
Modules

- **Python** has a way to put (function) definitions in a **file** and use them in a script or in an interactive instance of the interpreter. Such a file is called a **module**
  - *Official Python Documentation*
- Python's design philosophy encourages the use of **modules**
  - Python comes installed with official **modules**
  - There are thousands of community **modules**
  - Custom **modules** are easy to create



Module File

- ➔ **Modules** can contain function definitions and scripts
  - ◆ The **module's** Scripts are run when the module is imported
- ➔ Each **module** has its own space of variable names



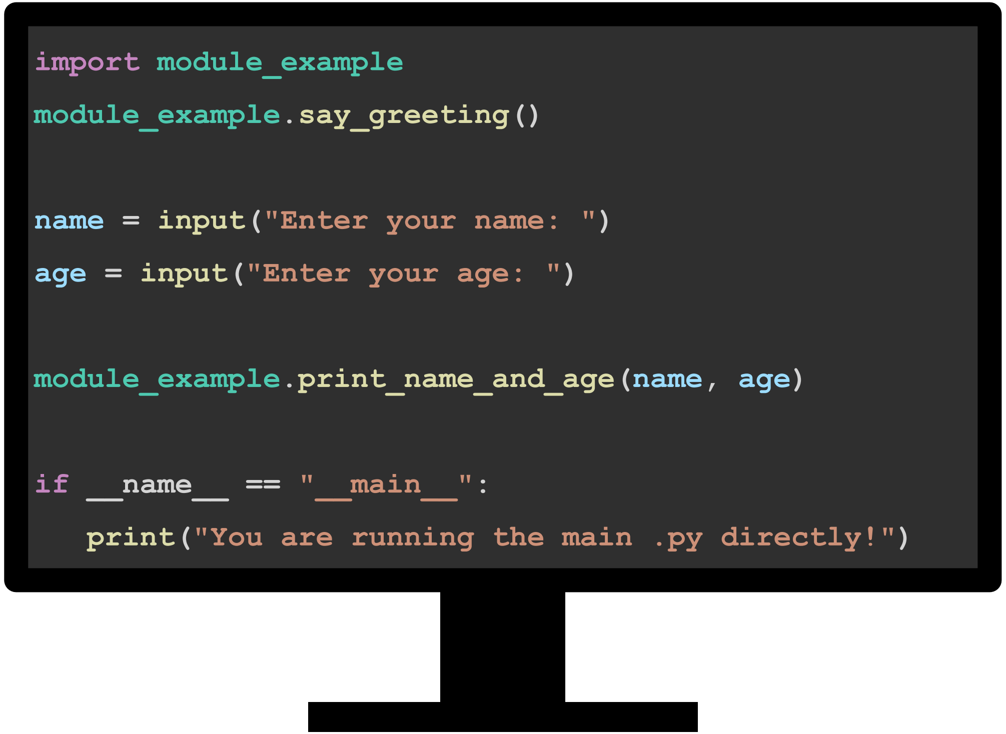
```python
def say_greeting():
   print("Hello!  Welcome to my Module!")

def print_name_and_age(name, age):
   print(f"Hello, {name}, you are {age}")
```

Where Will Python import from?

1. In the same directory
2. PYTHONPATH environment variable
3. Directories listed in installation
    a. Built-in Module
4. Dynamically adding a directory at runtime
    a. sys.path.append(<module path>)

Importing

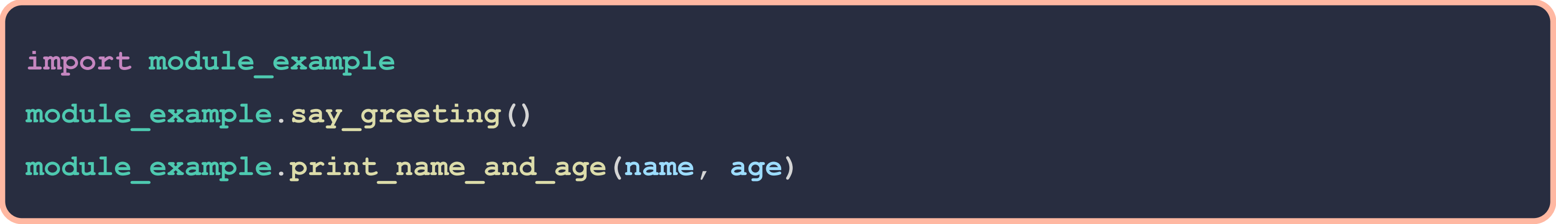
- Importing a **module** imports an object
    - **Not** the function definitions themselves
    - Module functions are methods on the imported object
    -
- **Modules** can import other **modules**



```
import module_example
module_example.say_greeting()


name = input("Enter your name: ")
age = input("Enter your age: ")


module_example.print_name_and_age(name, age)


if __name__ == "__main__":
    print("You are running the main .py directly!")
```

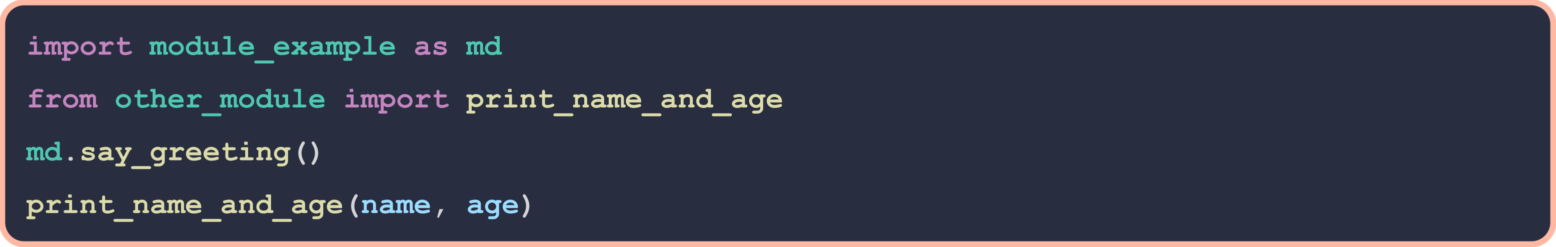
Importing

➔ Importing a **module** imports an object with the defined functions as methods



```
import module_example
module_example.say_greeting()
module_example.print_name_and_age(name, age)
```

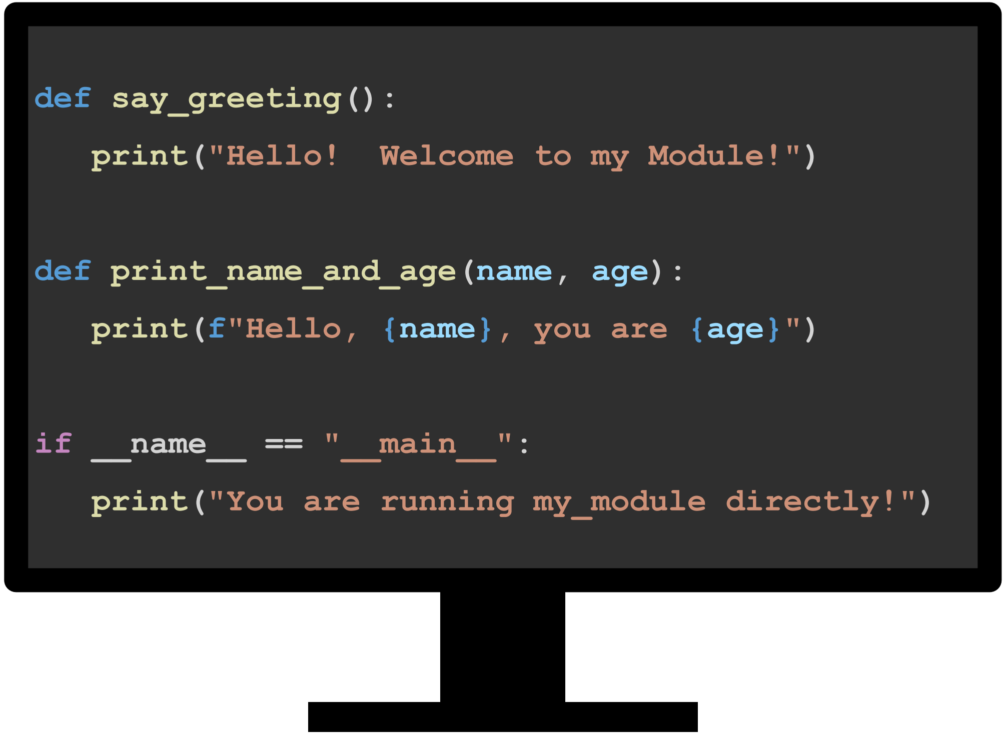
➔ Imports can be named, and specific imports can be selected



```
import module_example as md
from other_module import print_name_and_age
md.say_greeting()
print_name_and_age(name, age)
```

Module Script

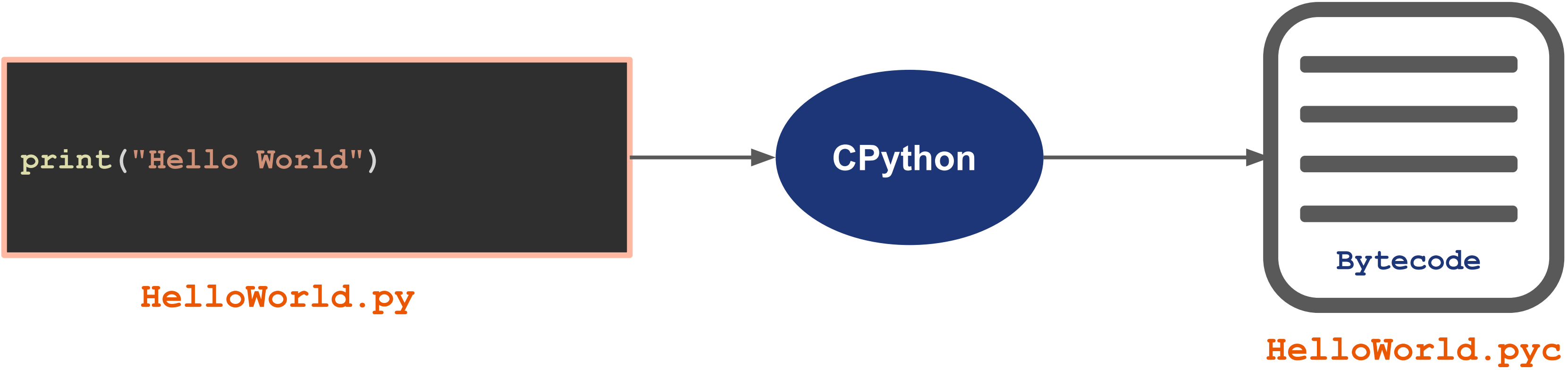
- **__name__** is a global value to the module
  - If the .py file is run directly, **__name__** is **“__main__”**
  - Otherwise, **__name__** is the name of the file
- Often used for development or testing
  - Test scripts can be called in the __main__



```
def say_greeting():
    print("Hello!  Welcome to my Module!")

def print_name_and_age(name, age):
    print(f"Hello, {name}, you are {age}")

if __name__ == "__main__":
    print("You are running my_module directly!")
```

“Compiled” Python Files

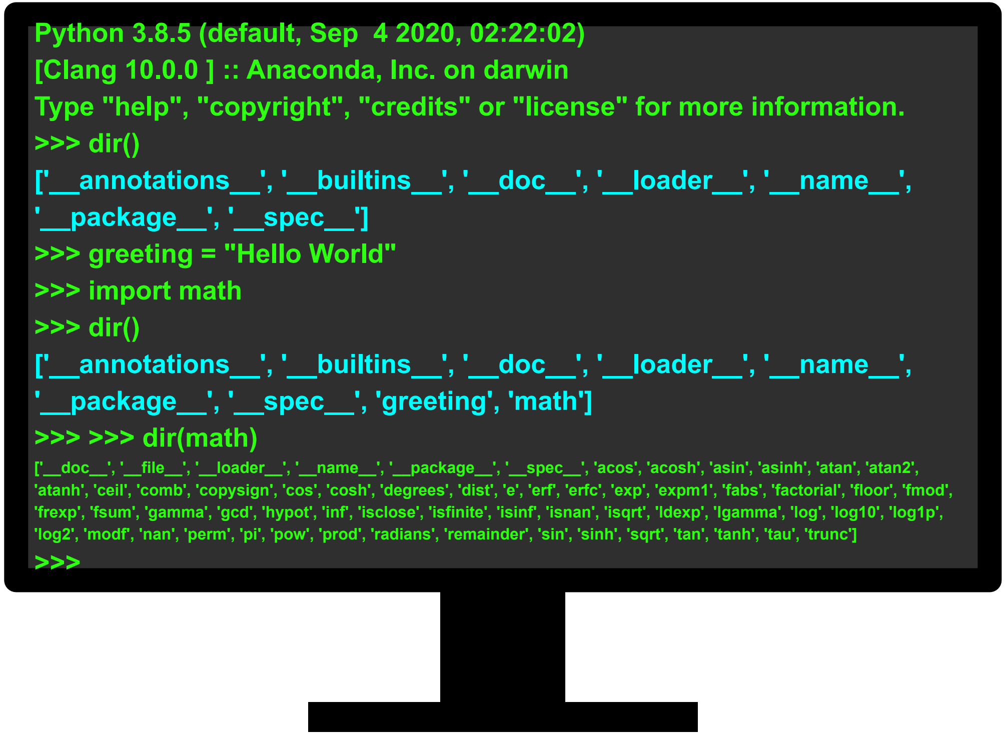
Python is an interpreted language, but it stores cached versions of interpreted modules in the __pycache__ directory for increased efficiency



```
print("Hello World")
```

dir() Function

- The **dir()** function will list available names
  - Variables and imports are names
- Passing an object to **dir(<object>)** will list the available attributes
- Passing a module **dir(<module>)** will list the available methods



```
Python 3.8.5 (default, Sep  4 2020, 02:22:02)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
>>> greeting = "Hello World"
>>> import math
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'greeting', 'math']
>>> >>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
```

Student Exercise

- Create a module to handle integer User input
  - All module functions implement try / except error handling
  - they should return the inputted value
  - Implement optional checking if a number is within a range
  - Allow the module caller to select the type of input the user enters
- Create a main script that will test the functions in the module