Python

A high-level, open-source, general programming language

Outline

1. Python History
2. Fundamentals and Syntax
3. Data Types and Operators
4. Collections
5. Conditionals and Control Structures
6. Exceptions
7. Functions
8. Classes and Objects
9. Files and OS Operations
10. Modules
11. Libraries

Prerequisites

1. https://www.python.org/downloads/
   - Download Python for your Operating System
2. https://code.visualstudio.com/
   - Visual Studio Code is the current standard for Integrated Development Environments
   - The **Python** and **Pylance** extensions are recommended
3. https://www.anaconda.com/products/individual
   - The Anaconda distribution provides a suite of tools for data science

Python Installation

Installation

- ➔ Windows
  - When installing via the download, be sure to add Python to the **environment variables**
  - Python can be installed via Chocolatey `choco install python`
  - A development only edition of Python is also available through the Windows store
- ➔ Mac OS
  - OS X comes with **Python 2.7** installed, you must install **Python 3**
  - Python 3 can also be installed through homebrew with the command `brew install python3`
- ➔ Linux
  - Many distributions of of Linux come with Python 3 already installed
  - The command `sudo apt-get install python3.6`can be used to install a specific version

The Python Language

Python History

- Python’s first deployment was in 1990
- Developed by the Python software Foundation
  - Centrum Wiskunde & Informatica (National Research Institute for Mathematics and Computer Science) in the Netherlands
- The principal author of the language was Guido van Rossum
- Python 2.0 was released in 2000
- Python 3.0 was released in 2008
  - It was not backwards compatible with Python 2

Python Attributes

1. Open Source
2. High Level
3. Interpreted
4. Object Oriented
5. Multi-Paradigm
6. Extensible

The Zen of Python

1: Beautiful is better than ugly	2: Explicit is better than implicit
3: Simple is better than complex	4: Complex is better than complicated
5: Flat is better than nested	6: Sparse is better than dense.
7: Readability counts	8: Special cases aren't special enough to break the rules
9: Although practicality beats purity	10: Errors should never pass silently
11: Unless explicitly silenced	12: In the face of ambiguity, refuse the temptation to guess
13: There should be one-- and preferably only one --obvious way to do it	14: Although that way may not be obvious at first unless you're Dutch
15: Now is better than never	16: Although never is often better than *right* now
17: If the implementation is hard to explain, it's a bad idea	18: If the implementation is easy to explain, it may be a good idea
19: Namespaces are one honking great idea -- let's do more of those!	20:

Python Pros and Cons

Advantages

- ➔ Easy to learn; naturalistic syntax
- ➔ Powerful; extensible with libraries
- ➔ Popular with an active community

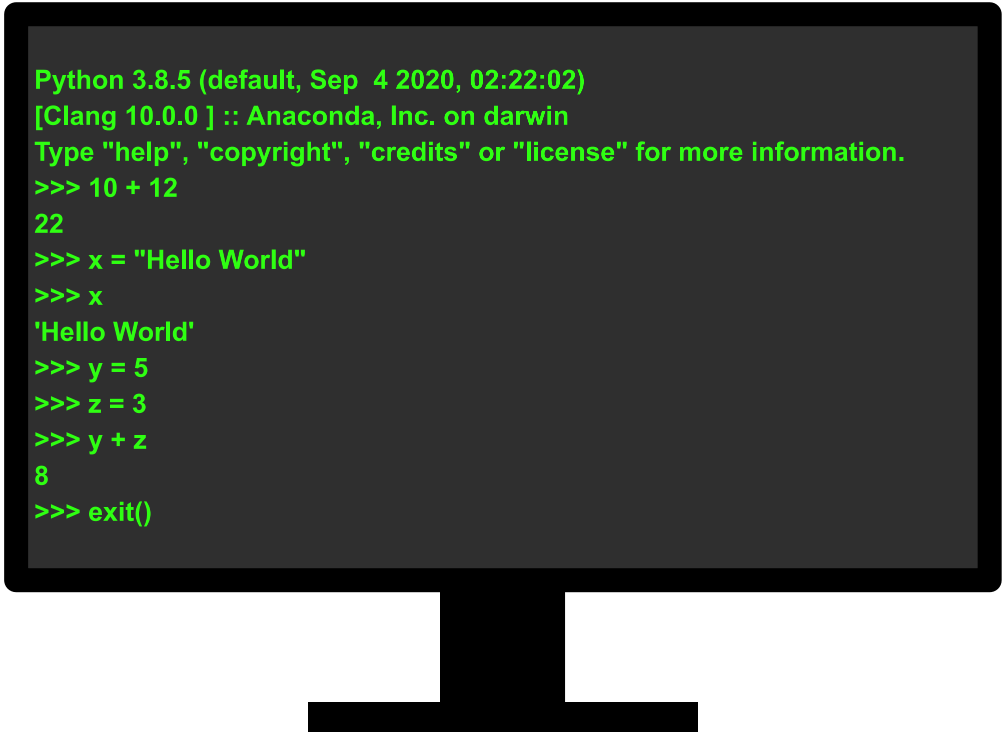
Disadvantages

- ➔ Interpetereted, not Compiled
- ➔ Comparatively high memory usage
- ➔ Dynamic Typing

Fundamentals and Syntax

Python Shell

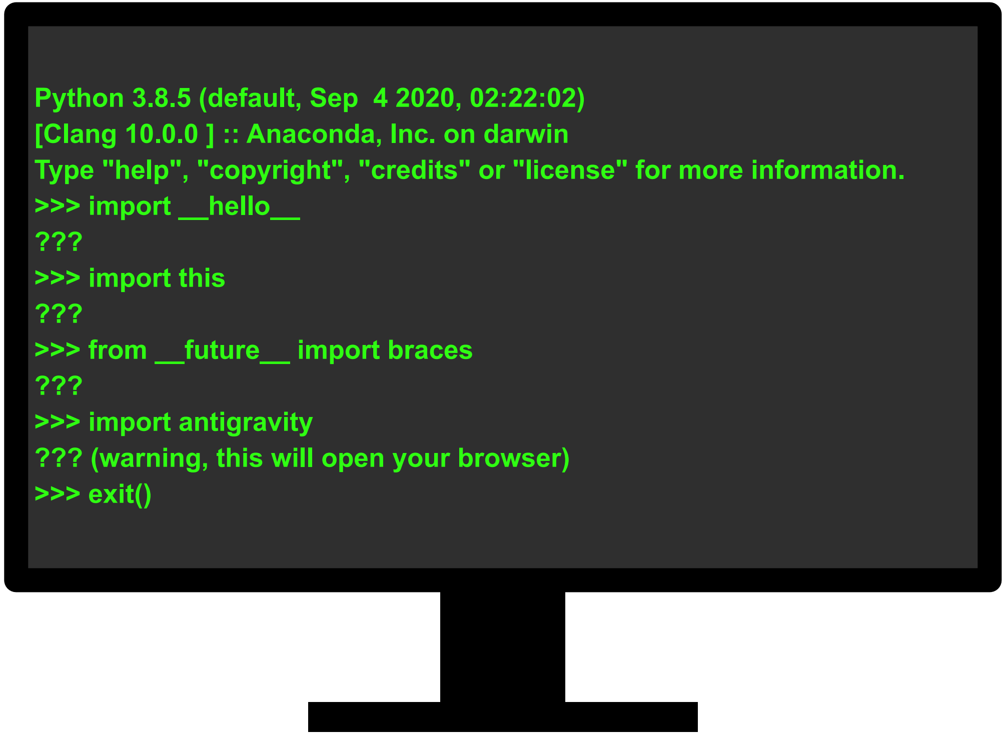
- ➔ To check your installation, enter one of the following commands:
  - ◆ `python --version`
  - ◆ `python3 --version`
- ➔ Enter the following command to enter the Python Shell
  - ◆ `python --version`
  - ◆ `python3 --version`
- ➔ Enter commands directly into the Command lIne
- ➔ The following command will exit
  - ◆ `exit()`



```
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.
>>> 10 + 12
22
>>> x = "Hello World"
>>> x
'Hello World'
>>> y = 5
>>> z = 3
>>> y + z
8
>>> exit()
```

Easter Eggs and Jokes

- ➔ Python was named after **Monty Python's Flying Circus**
  - ◆ Non-essential parts of the code contain references to several pop culture properties
- ➔ Here are some potentially amusing shell commands to try:
  - ◆ import __hello__
  - ◆ import this
  - ◆ from __future__ import braces
  - ◆ import antigravity
    - (minor warning, this one will open your browser)



```
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.
>>> import __hello__
???
>>> import this
???
>>> from __future__ import braces
???
>>> import antigravity
??? (warning, this will open your browser)
>>> exit()
```

.py Files

- ➔ Most Python development is done through modifying **.py** files
- ➔ Python code is executed sequentially, one instruction at a time
  - ◆ Functions are not hoisted
- ➔ A .py file can be executed from the **terminal** with
  - ◆ `python <filename>.py`
- ➔ **Jupyter Notebook** cells can be executed individually
- ➔ The **Spyder** environment allows you to execute specific lines

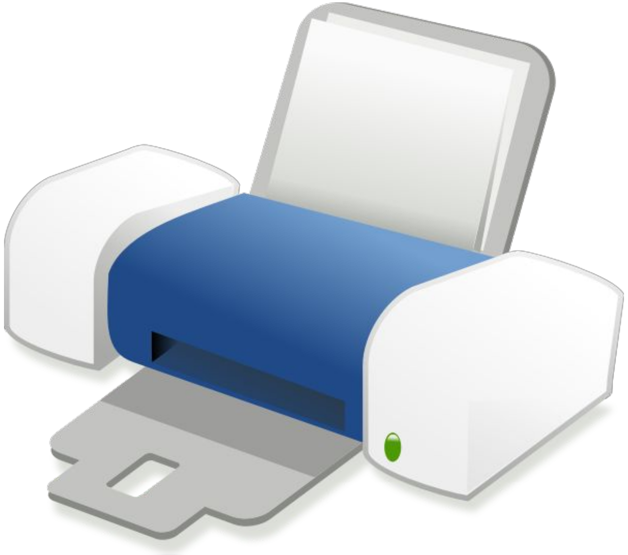
Comments

- ➔ Python Single line comments are declared with #

```
# Well commented code is essential any project
```

- ➔ Python does not officially support multi-line comments, but any strings not assigned to a variable will be skipped by the interpreter

```
"This line will be skipped by the interpreter"
"""
The triple quotation mark syntax starts a multi-line string
   They will preserve both indentation and line spacing
This is standard syntax for multi-line comments in Python
"""
```



The Print Function

- ➔ The print function will print a value to standard output
  - ◆ It is a globally available function

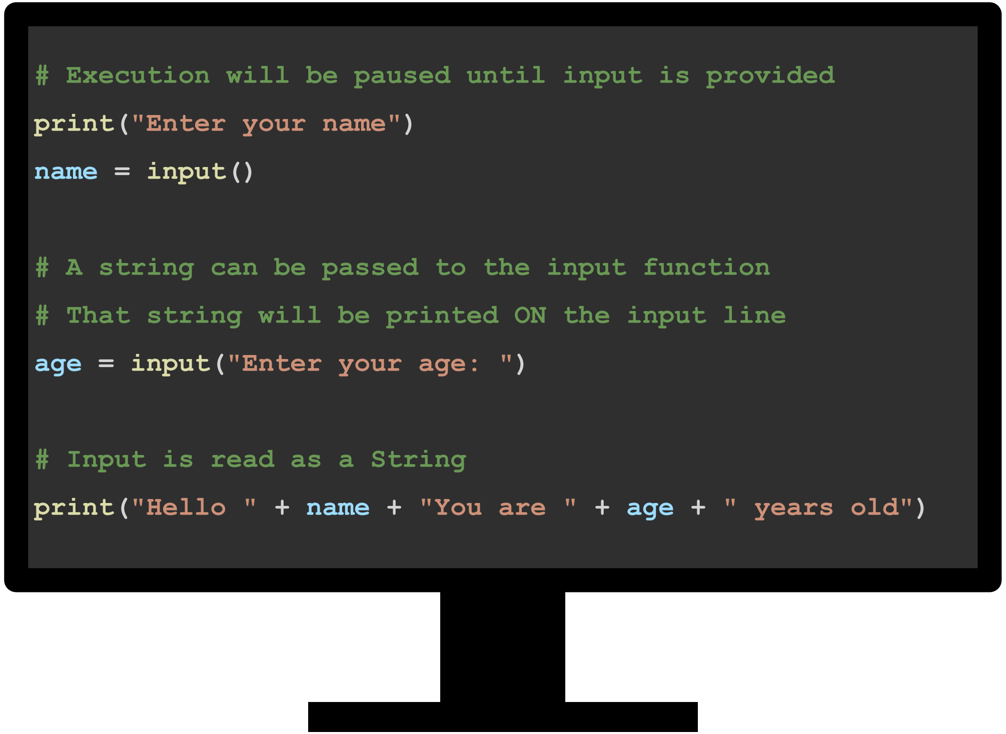
```
print("Hello World")
```

- ➔ Print can accept any data type
  - ◆ Data Types cannot be directly mixed

```
print (["Hello", "World"])
print (42)
# Will not work.
# print ("Hello World" + 42)
```

Input Function

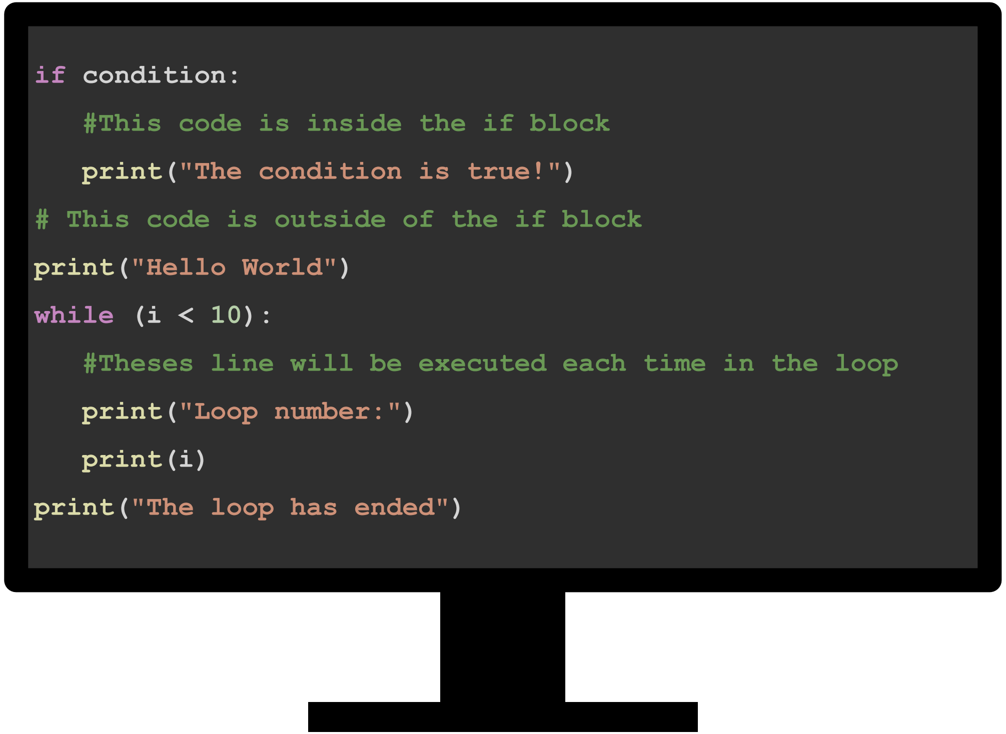
- ➔ The built-in input function will read from the standard input
- ➔ Characters are read as Strings
- ➔ The execution of the program will stop until input is entered



```
# Execution will be paused until input is provided
print("Enter your name")
name = input()

# A string can be passed to the input function
# That string will be printed ON the input line
age = input("Enter your age: ")

# Input is read as a String
print("Hello " + name + "You are " + age + " years old")
```

Indentation

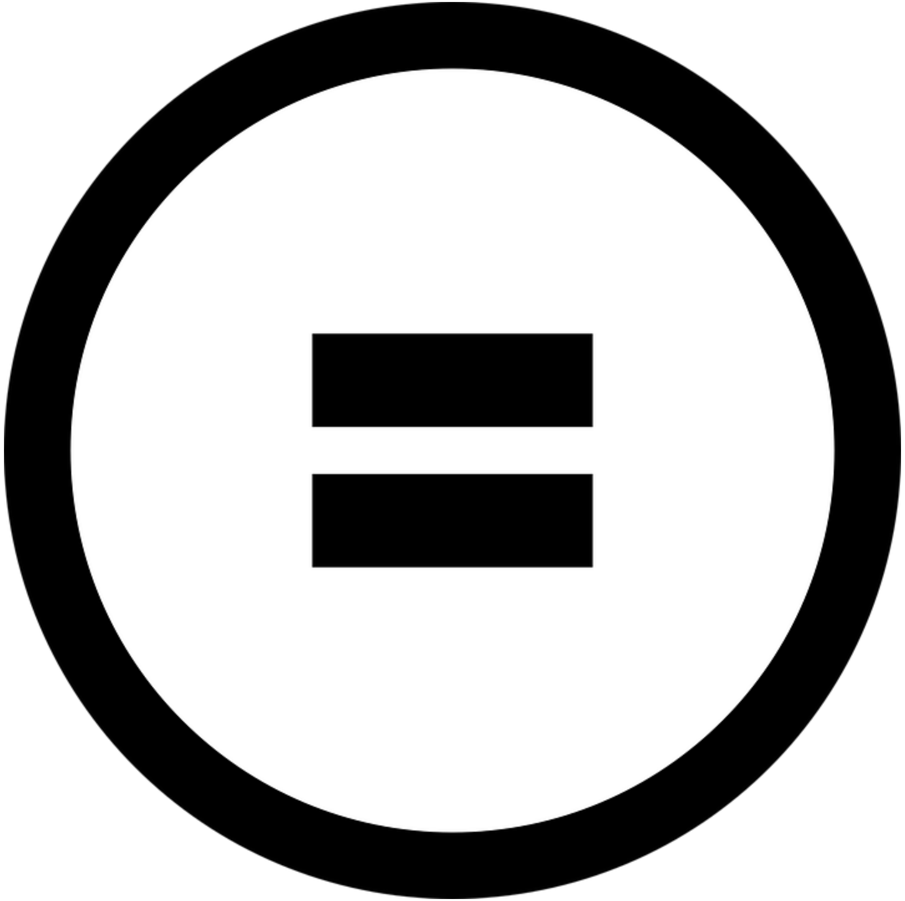
- ➔ Python uses **indentation** and whitepsace to define **code blocks**
  - ◆ Many languages use the curly brackets for this { }
- ➔ A colon : is used to instantiate a **block**
- ➔ The number of spaces per **indent** is up to the developer, but it must be consistent



```
if condition:
   #This code is inside the if block
   print("The condition is true!")
# This code is outside of the if block
print("Hello World")
while (i < 10):
   #Theses line will be executed each time in the loop
   print("Loop number:")
   print(i)
print("The loop has ended")
```

Student Exercise

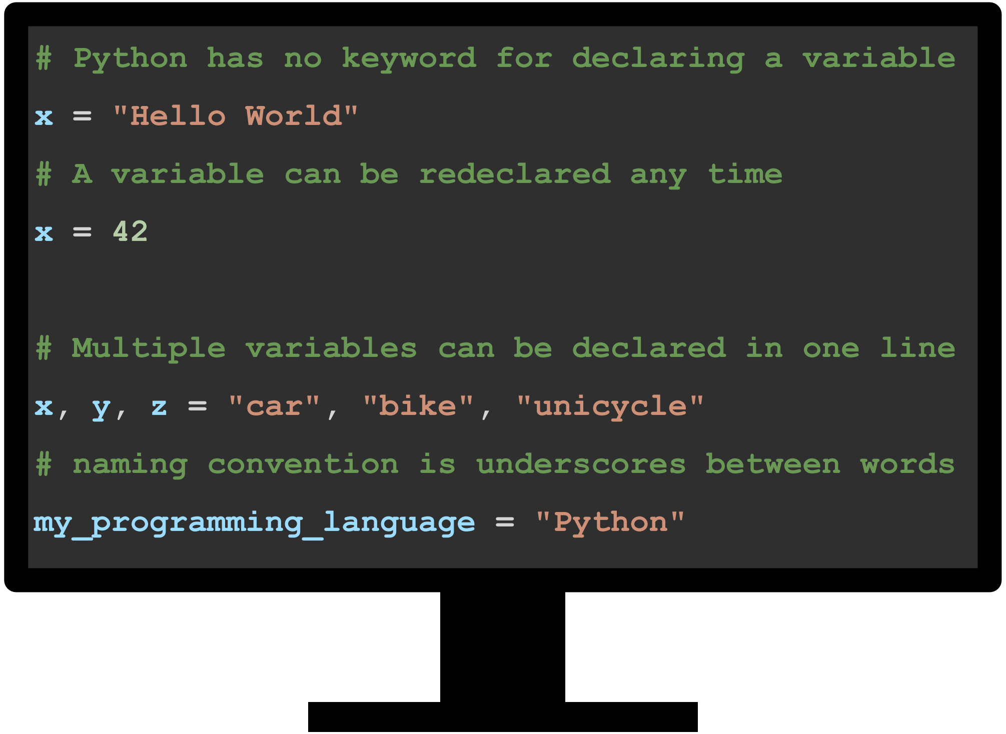
- Let’s make sure all of the installations worked properly!
- Create a .py file that accepts user input and formats an output
  - Prompt for the user for their name
  - Prompt the user for their age
  - Prompt the user for their profession
  - Print all of the information entered by the user in a single sentence



Variables and Data Types

Dynamic Typing

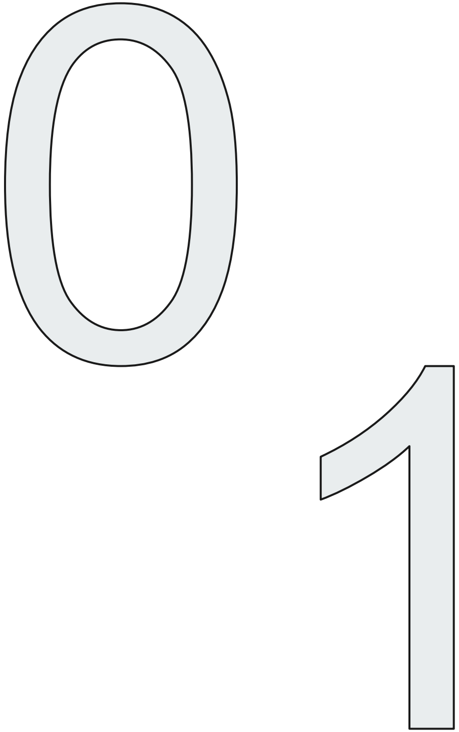
- Python is a **Dynamically-typed** language
  - The type of variable is not determined on **declaration**
  - A variable can be redeclared at any time to **any type**
  - Functions have no fixed return type or argument types
- There are extensions to Python to add static type checking
  - mypy



```
# Python has no keyword for declaring a variable
x = "Hello World"
# A variable can be redeclared any time
x = 42


# Multiple variables can be declared in one line
x, y, z = "car", "bike", "unicycle"
# naming convention is underscores between words
my_programming_language = "Python"
```

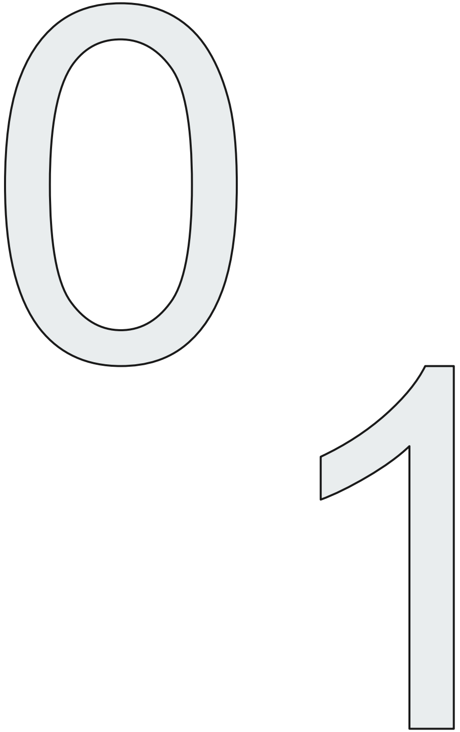
Booleans

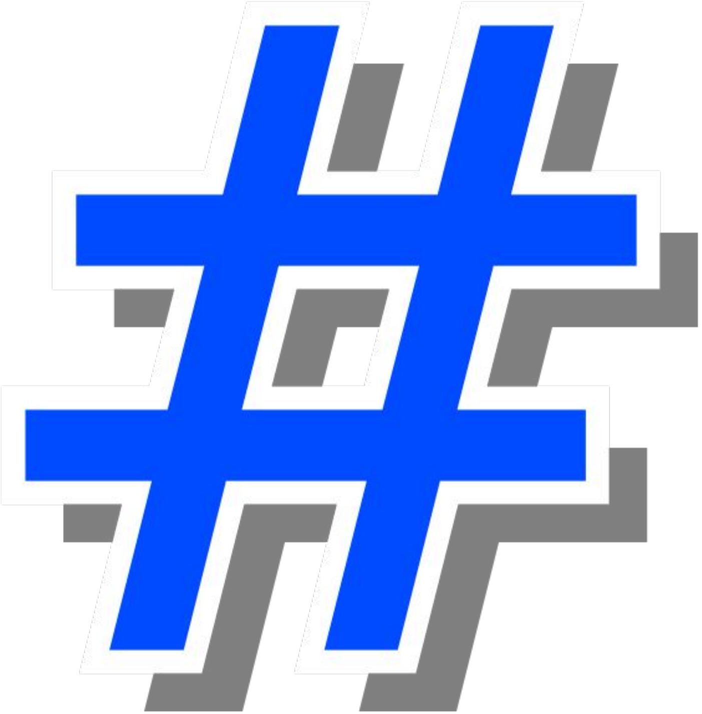
- Identified by **bool**
- Have a value of True or False
  - Boolean variables are capitalized in Python
- Logical operators resolve to Booleans



Booleans

- Objects with content resolve to **True**:
  - Any number other than **0**
  - sets ( ), lists [ ] , and dicts { } with content
  - Any String other than the empty String
  - The keyword **True**
- The following values resolve to **False**:
  - The number **0**,
  - Empty sets ( ), lists [ ] , and dicts { },
  - Empty strings “”
  - The keywords **None** and **False**





Numbers

- ➔ Python has 3 number types:
  - ◆ **int**: Whole numbers, positive or negative, of arbitrary length

```
x = 10
```

  - ◆ **float**: Positive or negative number containing one or more decimals

```
x = 37.43
```

  - ◆ **complex**: Numbers with an imaginary component, denoted with a j

```
x = 6j
```

- ➔ Python has a robust collection of mathematical features

Strings

- ➔ Identified by **str**
- ➔ Python considers strings as arrays of unicode characters
  - ◆ Strings support all List methods, as well as their own methods
- ➔ Python strings can be declared with single or double quotes

```
first_string = 'Hello'
other_string = "World"
```

- ➔ len()
  - ◆ Globally available function
  - ◆ Returns the length of a string

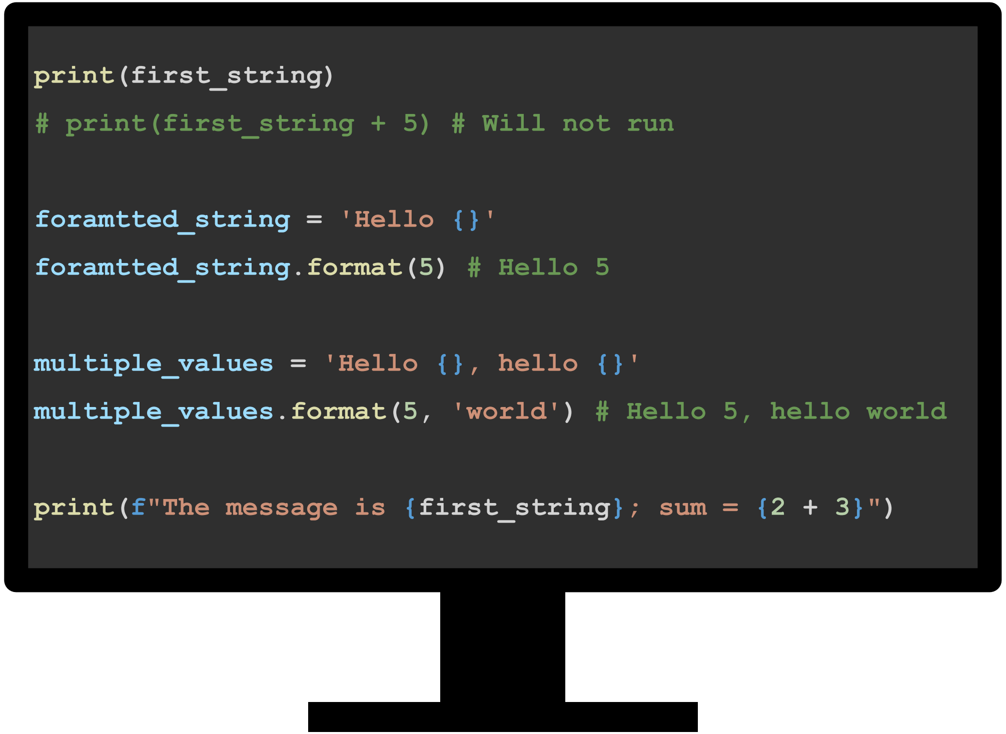
```
len("Hello World") # returns 11
```

String Methods

Method	Description
.capitalize()	Converts the first character to upper case
.count(str)	Returns the number of times a specified value occurs in a string
.find(str)	Searches the string for a specified value and returns the position of where it was found
.index(str)	Searches the string for a specified value and returns the position of where it was found
.join()	Joins the elements of an iterable to the end of the string
.replace(regex, str)	Returns a string where a specified value is replaced with a specified value
.split(regex)	Splits the string at the specified separator, and returns a list
.upper()	Converts a string into upper case

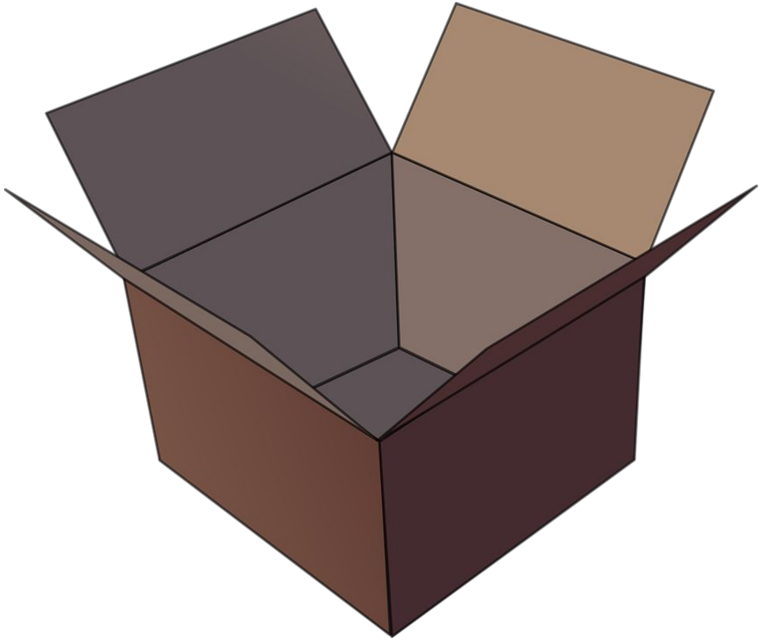
String Format

- **Strings** can be concatenated to other **Strings**, but not to other data types
- The The **.format()** method will add any character to a string in the curly braces
    - Multiple values can be added to a string
- The **f string** syntax automatically formats strings



```
print(first_string)
# print(first_string + 5) # Will not run

foramtted_string = 'Hello {}'
foramtted_string.format(5) # Hello 5

multiple_values = 'Hello {}, hello {}'
multiple_values.format(5, 'world') # Hello 5, hello world

print(f"The message is {first_string}; sum = {2 + 3}")
```

None



- ➔ **None** represents an empty value
  - ◆ This is different from 0, or the empty string
- ➔ **None** is an object, but has no methods
- ➔ All **None** values share the same object

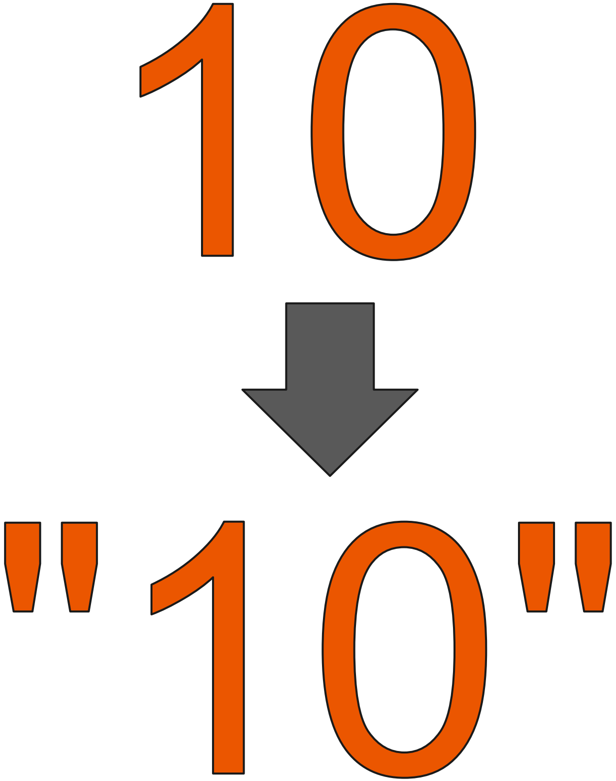
```
empty = None
```

Checking Data Types

- ➔ **type(variable)**
  - ◆ Globally available function
  - ◆ Returns the class of the type of object
  - ◆ **str**, **int**, **float**, **complex**, **bool**
- ➔ **isinstance(variable, class)**
  - ◆ Globally available function
  - ◆ Returns a boolean based on if the passed in variable is an instance of the given class

Casting Data Types

- **int()**
  - converts a string to an int
  - floors a float
- **float()**
  - add a .0 to a int
  - converts a string to a float
- **str()**
  - can take in a variety of arguments and converts them to a string
- If a variable is attempted to be cast to an incompatible data type, a **ValueError** is thrown



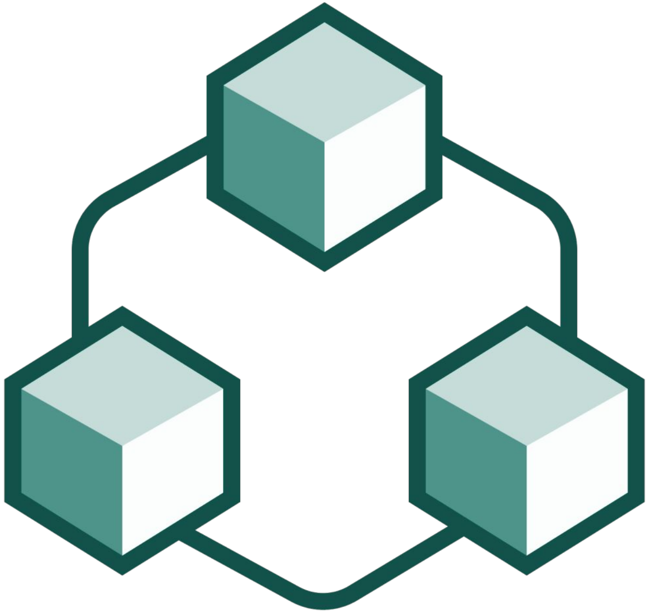
10

"10"

Student Exercise

- Write a program to capture user input for an Employee
- Record the employee’s name
    - Split the name into first and last
    - Make sure that the first letter of both is capitalized, and the rest of the letters are lowercase
- Record the employee’s age
    - Parse the age information to an int
- Generate the employee’s email
    - Concatenate the first and last names with a “.”
    - Add the employee’s id number to the last name
    - Add @company.com to the end
- Print all results to the screen

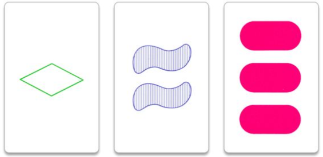
Python Collections



4 Collection Types

➔ List

- class list
- Ordered, Indexed, Mutable, allows duplicates

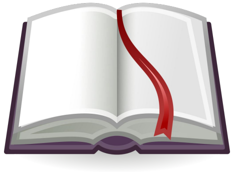


➔ Set

- class set
- Unordered, Unindexed, Mutable, does not allow duplicates

➔ Tuple

- class tuple
- Ordered, Indexed, Immutable, allows duplicates

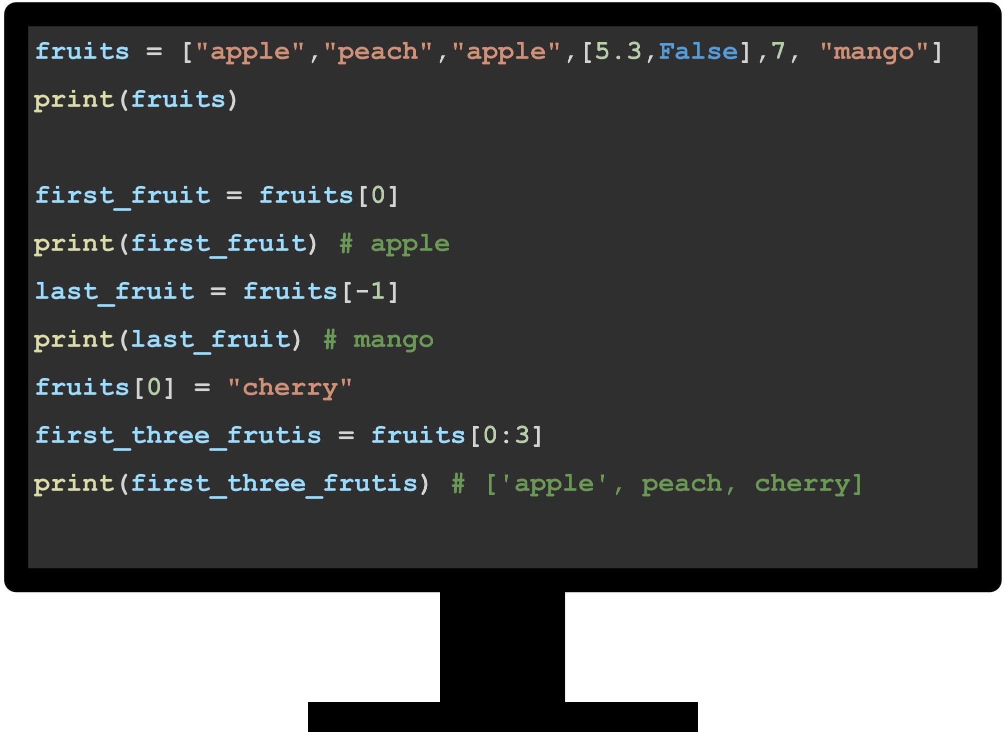


➔ Dictionary

- class dict
- Unordered, Key/Value Pairs, Mutable

List

- ➔ No **Array** type in Python
- ➔ Declared with square brackets [ ]
- ➔ Elements can be accessed with square brackets [index]
  - ◆ Indexes start at 0
  - ◆ Negative indexes indicate a selection from the end
  - ◆ Colon selects a range



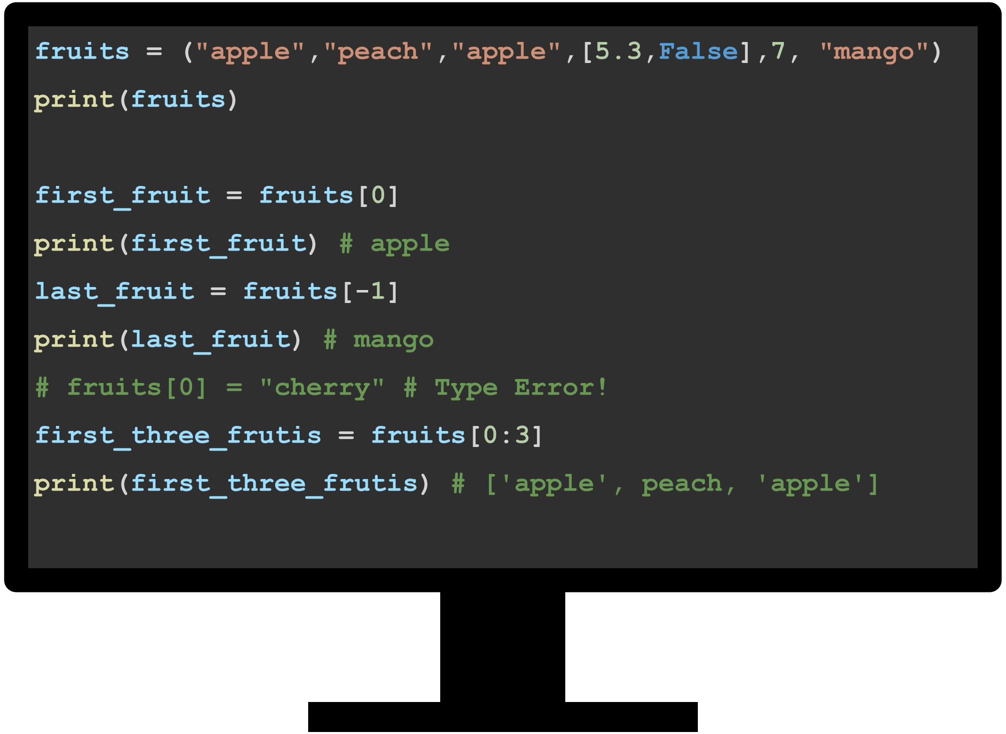
```
fruits = ["apple","peach","apple",[5.3,False],7, "mango"]
print(fruits)


first_fruit = fruits[0]
print(first_fruit) # apple
last_fruit = fruits[-1]
print(last_fruit) # mango
fruits[0] = "cherry"
first_three_frutis = fruits[0:3]
print(first_three_frutis) # ['apple', peach, cherry]
```

List Methods

Method	Description
.append(element)	Adds an element to the end of the list
.clear()	Removes all elements from the list
.copy()	Returns a copy of the list
.count(element)	Returns the number of that element in the list
.extend(iterable)	Adds all elements of the iterable to the list
.index(element)	Returns the first index of the element
.insert(index, elm)	Adds the given element at the given index
.pop(index)	Removes the element at the index and returns it
.remove(element)	Removes the first instance of the element
.reverse()	Reverses the order of the elements
.sort()	Sorts the list. Can be passed a sorting function

Tuple

- ➔ Declared with parenthesis **()**
- ➔ Elements can be accessed with square brackets **[index]**
  - ◆ Indexes start at 0
  - ◆ Negative indexes indicate a selection from the end
  - ◆ Colon selects a range



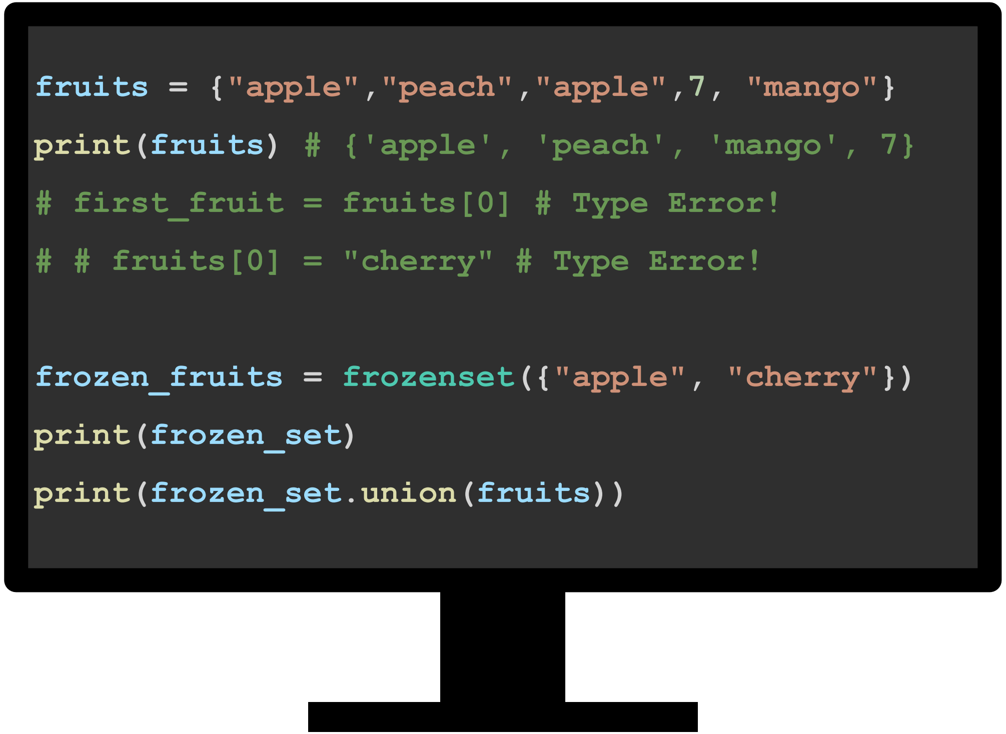
```
fruits = ("apple","peach","apple",[5.3,False],7, "mango")
print(fruits)

first_fruit = fruits[0]
print(first_fruit) # apple
last_fruit = fruits[-1]
print(last_fruit) # mango
# fruits[0] = "cherry" # Type Error!
first_three_frutis = fruits[0:3]
print(first_three_frutis) # ['apple', peach, 'apple']
```

Tuple Methods

Method	Description
.count(element)	Returns the number of that element in the list
.index(element)	Returns the first index of the element
+ operator	Tuples can be added to return a new tuple

Set

- ➔ Declared with curly brackets and at least one element {<element>}
  - ◆ Calling set() will create an empty set
- ➔ Elements cannot be individually accessed
  - ◆ Must be iterated over to access elements
  - ◆ Duplicates are not added
- ➔ frozenset is an immutable set



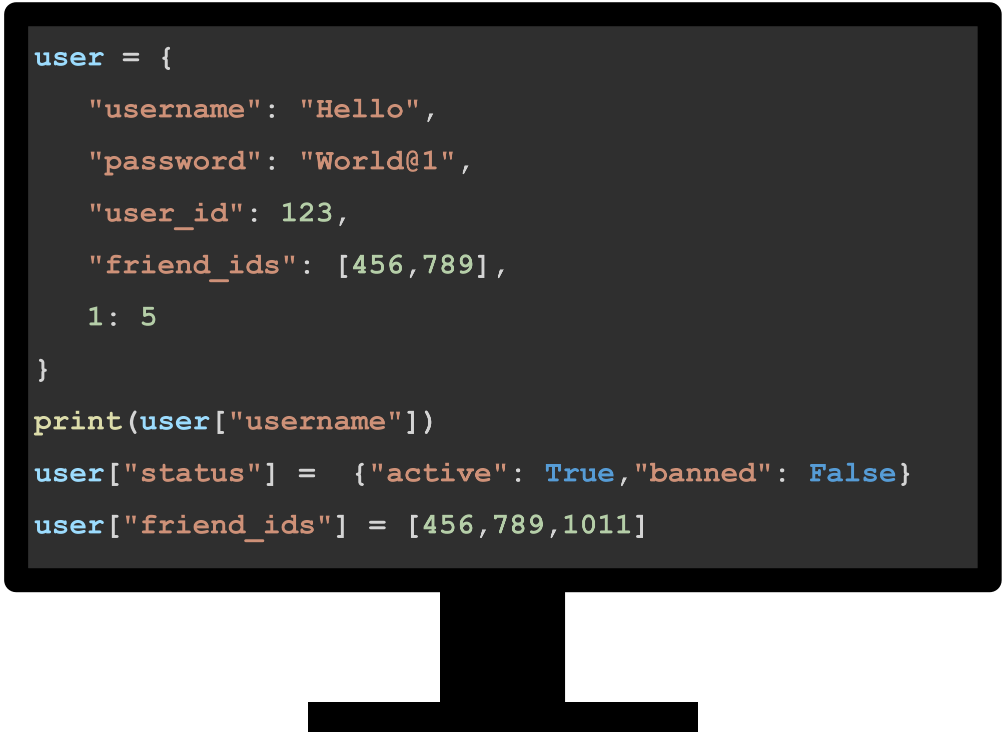
```
fruits = {"apple","peach","apple",7, "mango"}
print(fruits) # {'apple', 'peach', 'mango', 7}
# first_fruit = fruits[0] # Type Error!
# # fruits[0] = "cherry" # Type Error!


frozen_fruits = frozenset({"apple", "cherry"})
print(frozen_set)
print(frozen_set.union(fruits))
```

Set Methods

Method	Description
.add(element)	Adds an element to the end of the set
.difference(set)	Returns a set containing the difference between sets
.discard(element)	Removes the element from the set
.intersection(set)	Returns a set that is the intersection of the sets
.isdisjoint(set)	Returns whether two sets have a intersection or not
.issubset(set),	Returns whether the set is a subset of the given set
issuperset(set)	Returns whether the set is a superset of the given set
.pop()	Removes a random element and returns it
.remove(element)	Removes the given element and returns it
.union(set)	Returns the union of the two sets
.update(set)	Updates the set with a union of the given set

Dictionary

- Declared with curly brackets { }
    - Keys must be strings or numbers
- Elements can be accessed by their keys with the square brackets [ ]
    - Can iterated over to access elements
    - Duplicate keys update value



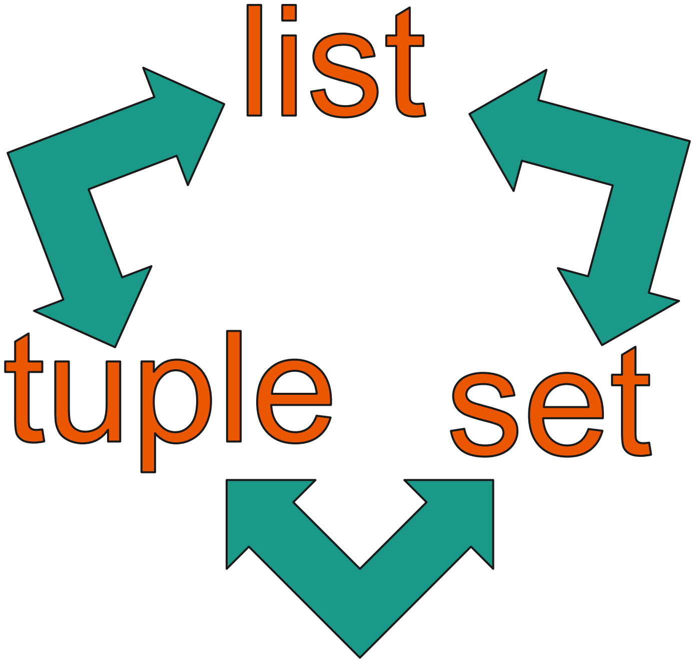
```
user = {
    "username": "Hello",
    "password": "World@1",
    "user_id": 123,
    "friend_ids": [456,789],
    1: 5
}
print(user["username"])
user["status"] =  {"active": True,"banned": False}
user["friend_ids"] = [456,789,1011]
```

Dictionary Methods

Method	Description
.clear()	Removes all elements from the dictionary
.copy()	Returns a copy of the dictionary
.fromkeys(iter, val)	Returns a dictionary with the listed keys and one value
.get(key)	Returns the value for the given key
.items()	Returns a list of tuples for each key/value pair
.keys(),	Returns a list of the dictionary’s keys
.pop(key)	Removes the value at the given key and returns it
.popitem()	Removes the last inserted value and returns it
.setdefault(key, val)	Returns the value of the key, or creates it
.update(dictionary)	Updates the dictionary with the given key/value
.values()	Returns a list of all values in the dictionary

Casing Between Collections

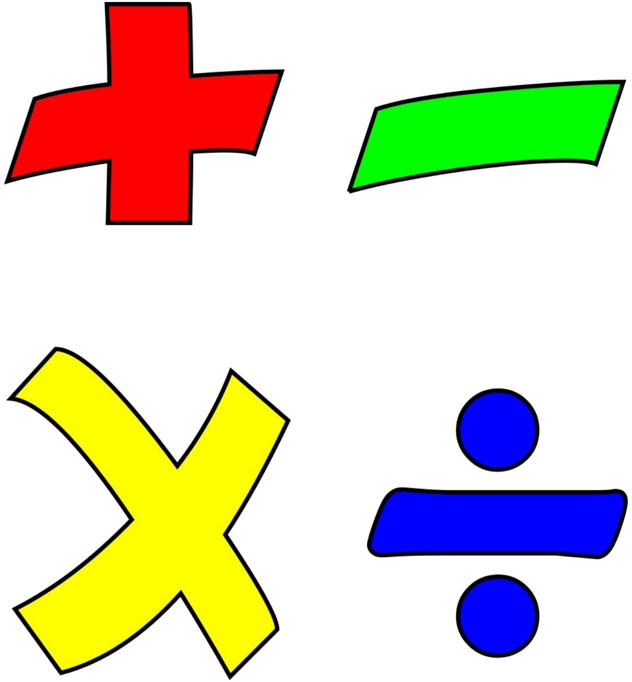
- Python can easily switch between one data structure an another
- set()
  - can turn a list or tuple into a set
- tuple()
  - can turn a set or list into a tuple
- list()
  - can turn a set or tuple into a list
- Casting from a tuple to a list can allow for altering elements, then casting it back to a tuple
- Casting from a set to a list can allow for indexing elements



Student Exercise

- Create a **dictionary** with "name", "age", and "years" keys
- Prompt the user to enter their first three programming languages
  - store them as a **tuple**
- Prompt the user to enter their three favorite programming languages
  - store them as a **list**
- Create a **set** that is a **intersection** of their first programming languages and their favorite programming languages
- Add all of these collections as keys to the dictionary you created
  - format a print statement to print the relevant data to the console

Python Operators



Arithmetic Operators

Operator	Name	Example
+	Addition	9 + 5 # 14
-	Subtraction	8 - 3 # 5
*	Multiplication	5 * 7 # 35
/	Division	4 / 2 # 2
%	Modulus	7 % 3 # 2
**	Exponential	5 ** 3 # 125
//	Floor Division	9 // 4 # 2

Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero Fill Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Assignment Operators

Operator	Name	Example
=	Assignment	x = 42 # 42

+=, -=, *=, /=, %=, **=, //=	Performs a mathematical operation then assigns a variable	y = 10
		y += 7
		# 17

&=, \=, ^=, >>=, <<=	Performs a bitwise operation then assigns the variable	z = 5 z &= 3 # 1

Comparison Operators

Operator	Name	Example of True
==	Equal To	10 == 10
!=	Not Equals	10 != 11
>	Greater Than	10 > 9
<	Less Than	10 < 20
>=	Greater Than or Equal To	10 >= 10
<=	Less Than or Equal To	10 <= 10

Logical Operators

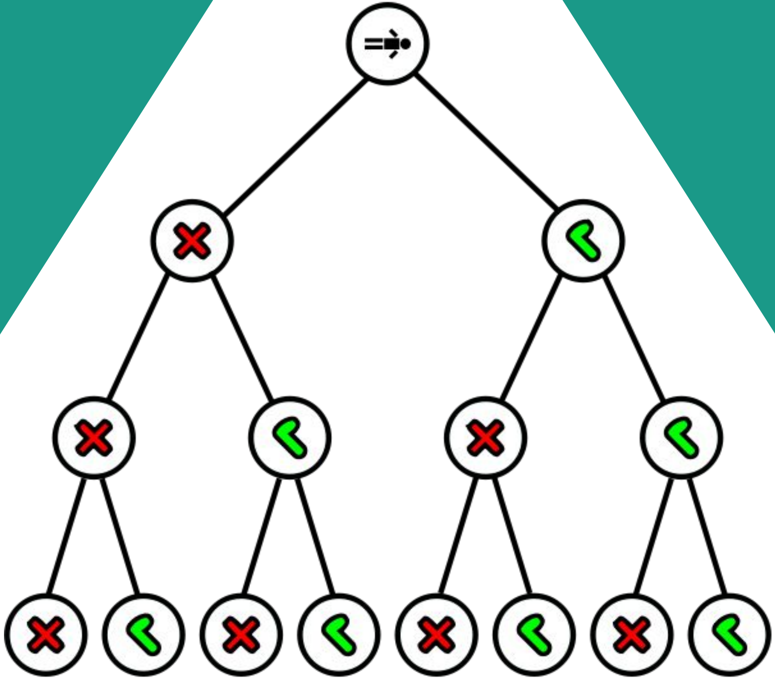
Operator	Description	Example of True
`and`	Returns True if both expressions are true	True `and` True
`or`	Returns True if at least one expression is true	True `or` False
`not`	Reverses a booleans	`not` (False)

Identity and Membership Operators

Operator	Description	Example of True
`is`	Returns True if both variables are the same object	“apple” `is` “apple”
`is not`	Returns True if both variables are not the same object	“apple” `is not` [1,2,3]

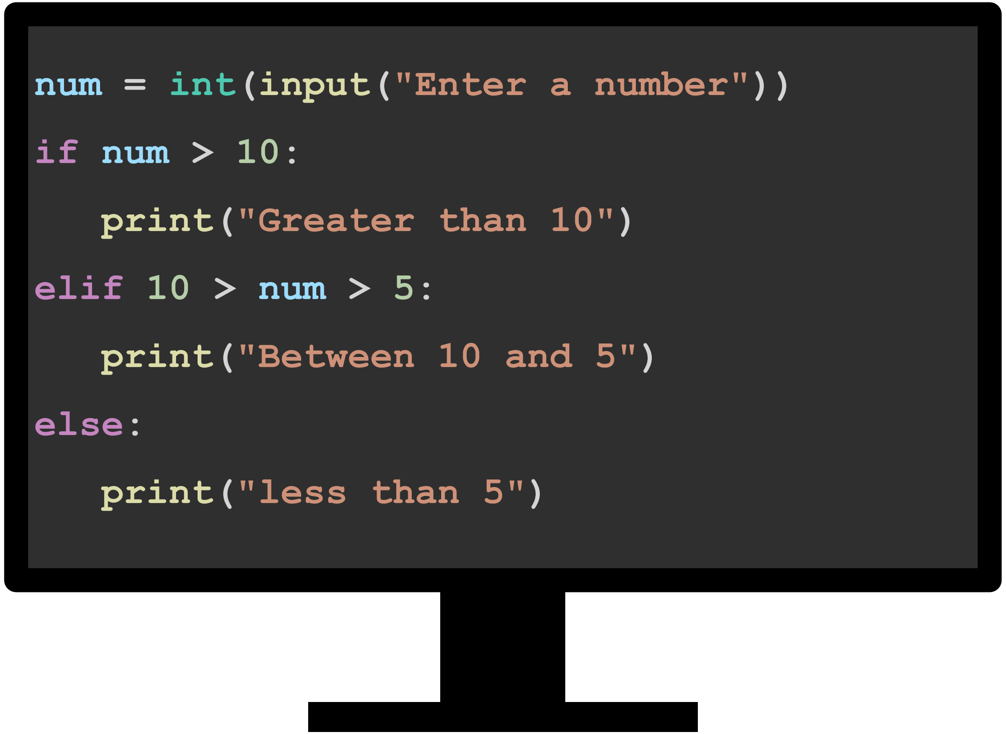
Operator	Description	Example of True
`in`	Returns True if a sequence with the specified value is present in the object	3 `in` [1,2,3]
`not in`	Returns True if a sequence with the specified value is not present in the object	5 `not in` [1,2,3]

Conditionals



if/elif/else

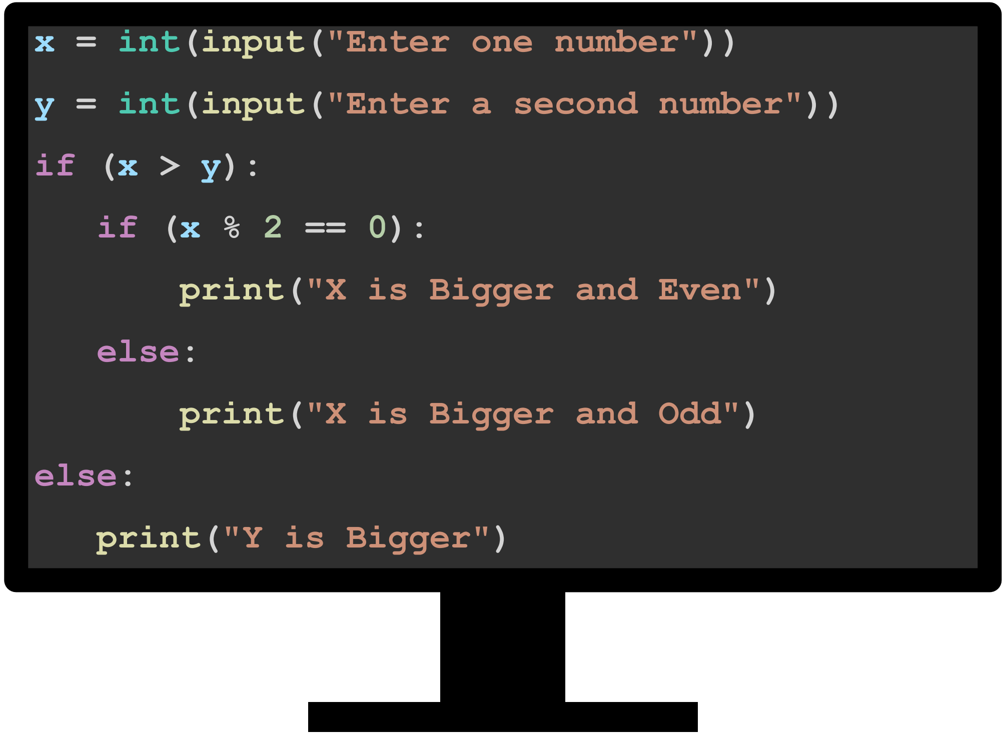
- ➔ Python’s Else If syntax is called **elif**
- ➔ Any condition that resolved to True will cause all statements within the if block to execute.
  - ◆ Remember that consistent indentation counts as a block



```python
num = int(input("Enter a number"))
if num > 10:
    print("Greater than 10")
elif 10 > num > 5:
    print("Between 10 and 5")
else:
    print("less than 5")
```

Nesting ifs

- ➔ Each indented block is considered its own **block**
- ➔ If statements can be nested within other if statements to chain conditionals



```
x = int(input("Enter one number"))
y = int(input("Enter a second number"))
if (x > y):
    if (x % 2 == 0):
        print("X is Bigger and Even")
    else:
        print("X is Bigger and Odd")
else:
    print("Y is Bigger")
```

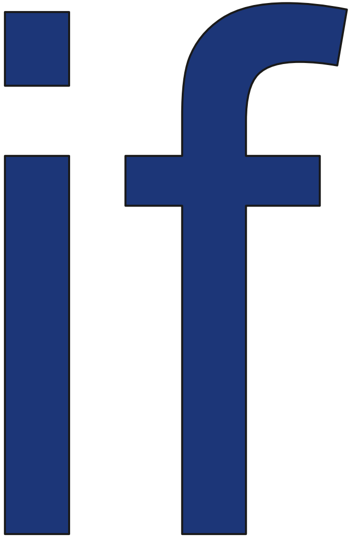
Single Line If

- ➔ A single line conditional does not need an indent

```
if x > y: print("x is greater than y")
elif x < y: print("y is greater than x")
else: print("x is equal to y")
```

- ➔ **Ternary Operators** set a value based on a condition
  - ◆ Often known as **Conditional Expressions**

```
message = "Greater than 10" if  (num > 10) else "Less than 10"
```



Student Exercise

- **Rock - Paper - Scissors**
  - Write a program that accepts the user’s input, and make sure that input is either **r**, **p**, or **s**
  - Ask for a second user’s input, and make sure that input is also only **r**, **p**, or **s**
  - Complete the program to output the results of a rock, paper, scissors game
  - The game only needs to run once

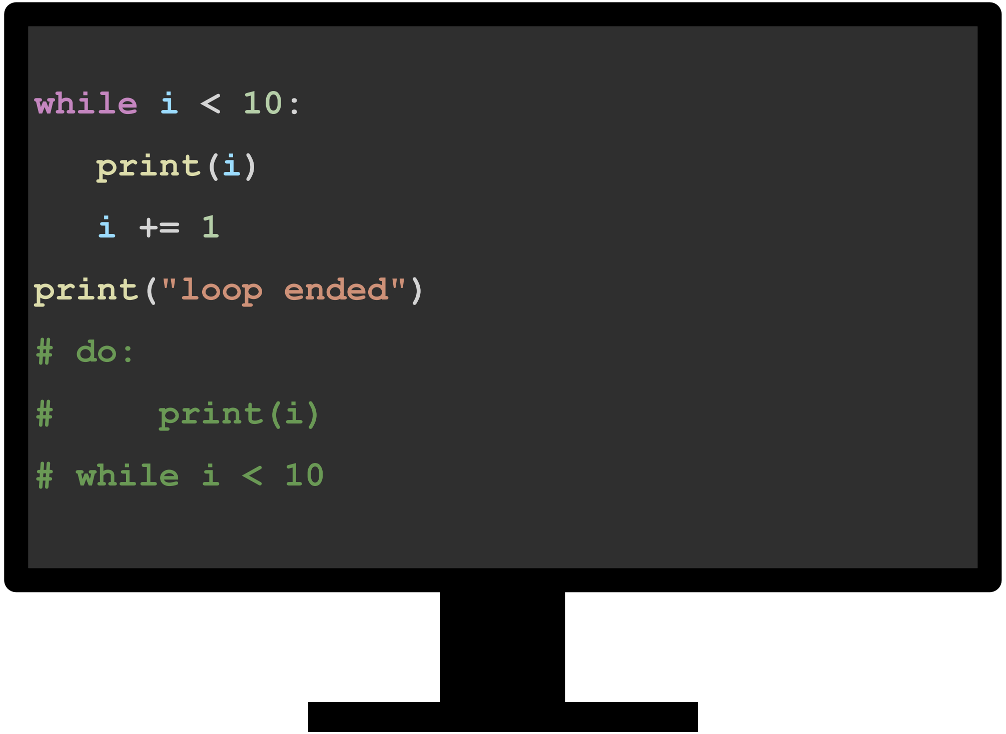
First known representation of the ouroboros on one of the shrines enclosing the sarcophagus of Tutankhamun

- Wikipedia

Python Loops

While Loop

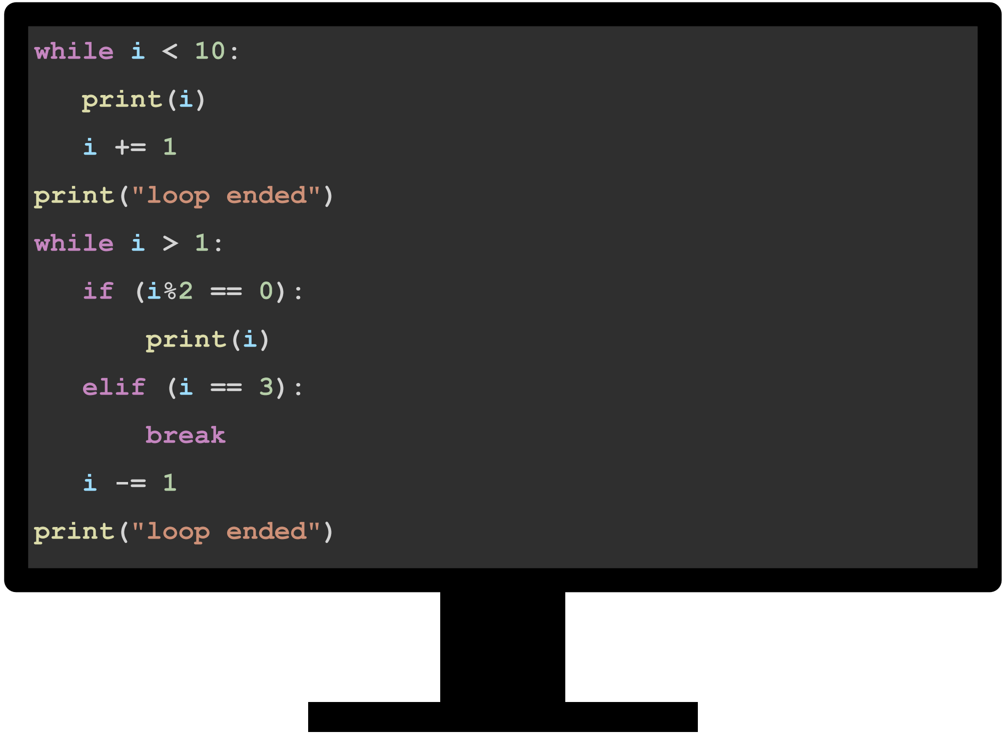
- **While Loops** execute a code block repeatedly until a given condition is met
  - The conditional must resolve to a boolean
  - **While Loops** do not have an internal counter
  - Python does not have a ++ increment operator
  - Python does **not** have a **do while** loop



```
while i < 10:
    print(i)
    i += 1
print("loop ended")
# do:
#     print(i)
# while i < 10
```

Break, Continue

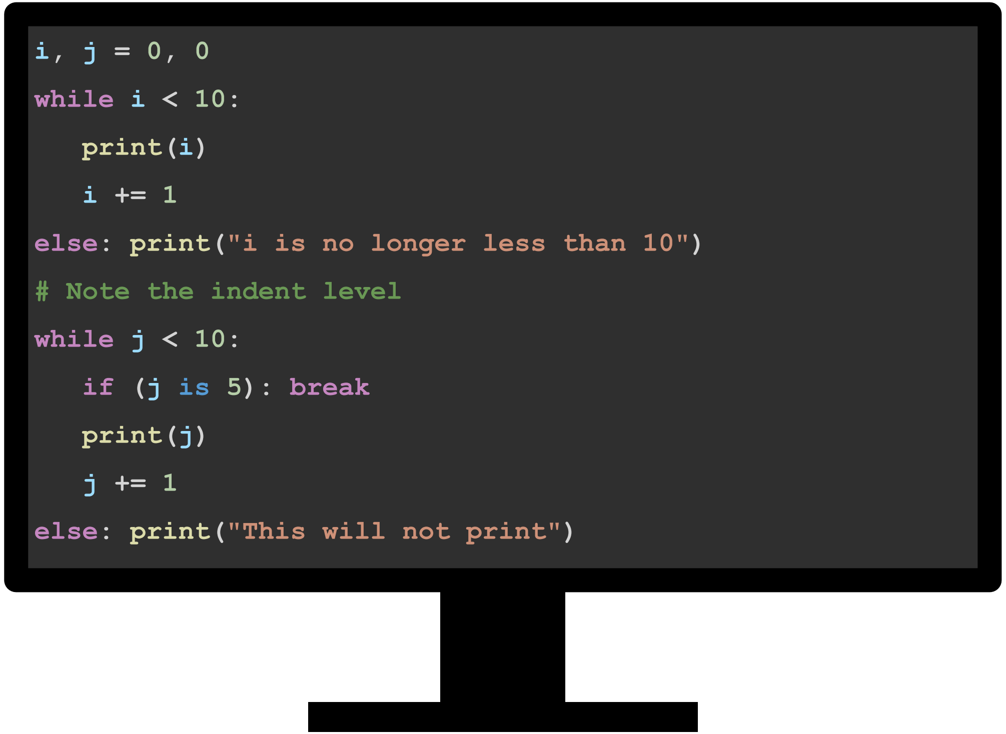
- ➔ A **break** statement will end a loop immediately
- ➔ A **continue** statement will jump to the next iteration of the loop



```
while i < 10:
    print(i)
    i += 1
print("loop ended")
while i > 1:
    if (i%2 == 0):
        print(i)
    elif (i == 3):
        break
    i -= 1
print("loop ended")
```

While … else

- ➔ The **Else clause** on a **while loop** will execute when a loop is finished
- ➔ If the loop ends before the **while** condition is met, the **else clause** will not execute



```
i, j = 0, 0
while i < 10:
   print(i)
   i += 1
else: print("i is no longer less than 10")
# Note the indent level
while j < 10:
   if (j is 5): break
   print(j)
   j += 1
else: print("This will not print")
```

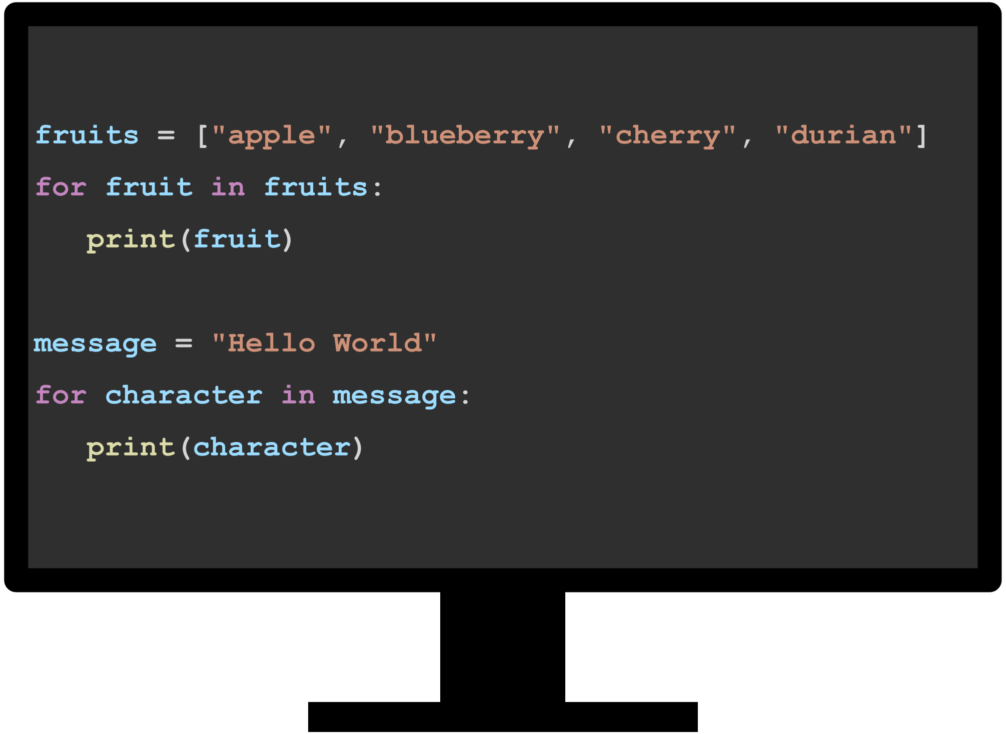
Iterators

- In Python, an iterator is any object that contains a countable number of items
  - **Lists**, **Dictionaries**, **Tuples**, and **Set** are all iterable objects
  - **Strings**, as character lists, are also iterable
- An iterator is an object that contains the following methods:
  - **__iter__()**: creates an iterable list out of the object
  - **__next__():** advances the iterable in one direction
- To generate an iterator, call the iter() method on an iterable object
  - **iter(my_list)**
  - **iter("Hello World")**

NEXT

For Construct

- ➔ **For Loops** in Python will execute over any sequence object
  - ◆ This is slightly different from other programming languages
- ➔ **For loops** can be created over any iterable object in Python
- ➔ The defined code block will act over each element in the iterable
  - ◆ The created variable will take the next value in the iterable



```python
fruits = ["apple", "blueberry", "cherry", "durian"]
for fruit in fruits:
    print(fruit)

message = "Hello World"
for character in message:
    print(character)
```

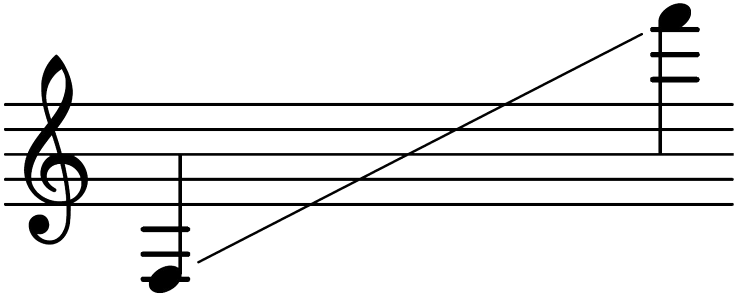
The Range Function

- The **range( )** function returns an iterable of numbers
  - It is a globally available function
  - By default, it starts at **0**, increments by **1** and ends before the argument number

```
my_range = range(10)# range from 0 - 9
```

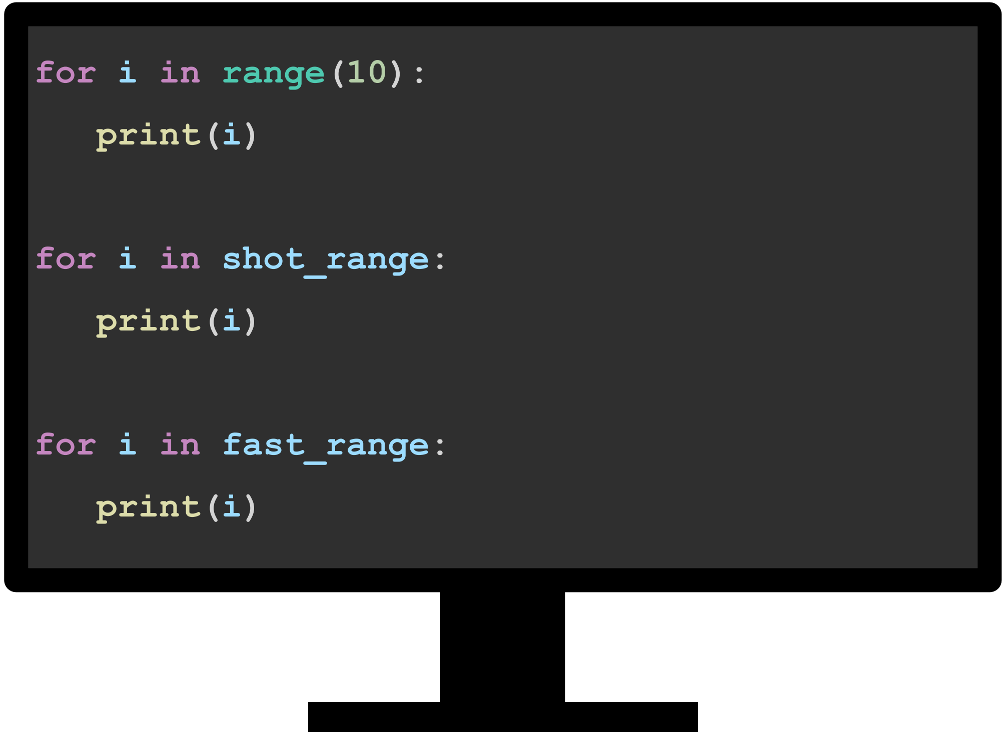
- Passing in a **first** parameter will set the **starting number**
- Passing in a **third** parameter will set the **increment**

```
# range from 3 - 9
shot_range = range(3, 10)
# range from 1 - 9, counting by 2
fast_range = range(1, 10, 2)
```



For … range( )

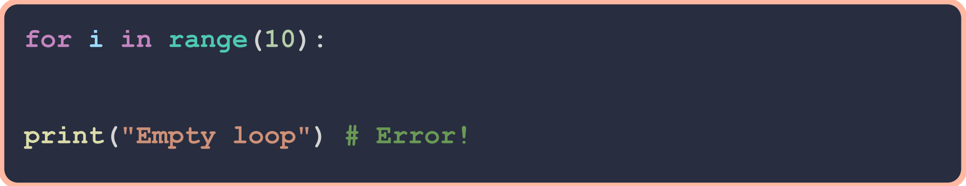
- ➔ **For Loops** can operate over a range()
  - ◆ **range()** returns an **iterator**
- ➔ This will replicate the traditional effect of a for(increment) loop



```
for i in range(10):
    print(i)

for i in shot_range:
    print(i)

for i in fast_range:
    print(i)
```

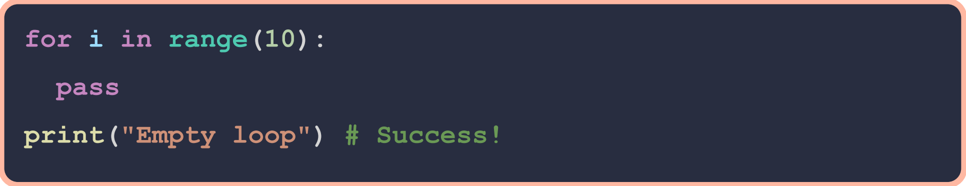
Pass Statement

- ➔ Python code blocks cannot be empty



```python
for i in range(10):

print("Empty loop") # Error!
```

- ➔ The **pass** statement will allow an empty block to be skipped



```python
for i in range(10):
  pass
print("Empty loop") # Success!
```

Student Exercise

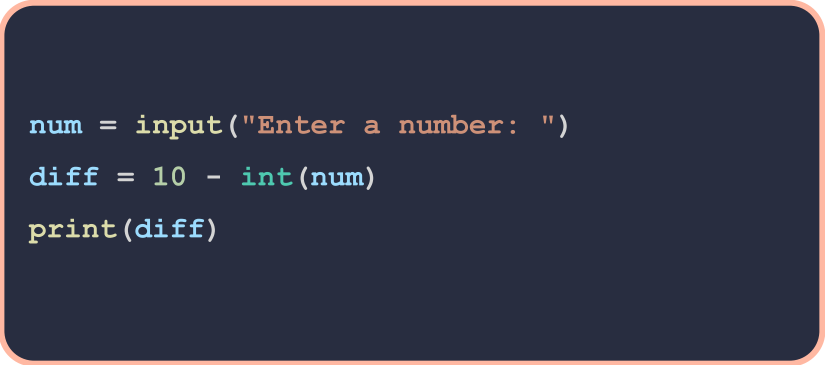
- ➔ FizzBuzz!
- ➔ Write a program that accepts a user's integer input
- ➔ For every integer between 0 and that number, add the following to a list:
  - ◆ If the number is divisible by **3**, add "**Fizz**"
  - ◆ If the number if divisible by **5**, add "**Buzz**"
  - ◆ If the number is divisible by both **3** and **5**, add "**Fizzbuzz**"
  - ◆ If the number is divisible by neither, add the number itself
  - ◆ Loop over the list and print each element in that list, then print the **sum** of all **integers**, and the count of **Fizz**, **Buzz** and **Fizzbuzz**

ERROR HANDLING

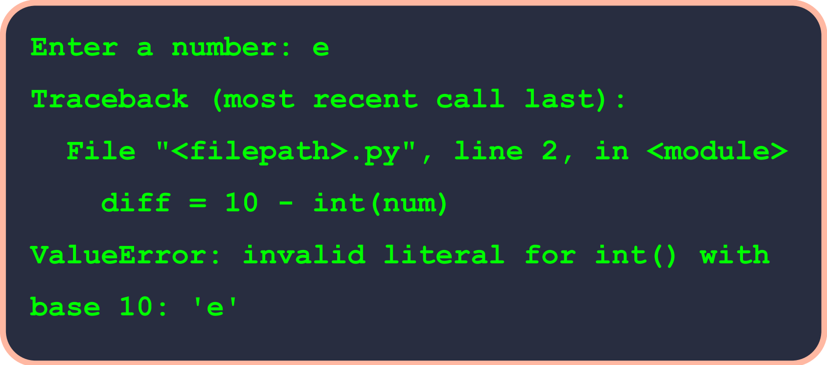
The Zen Of Python Principals
10 and 11...

Error Handling

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



```
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

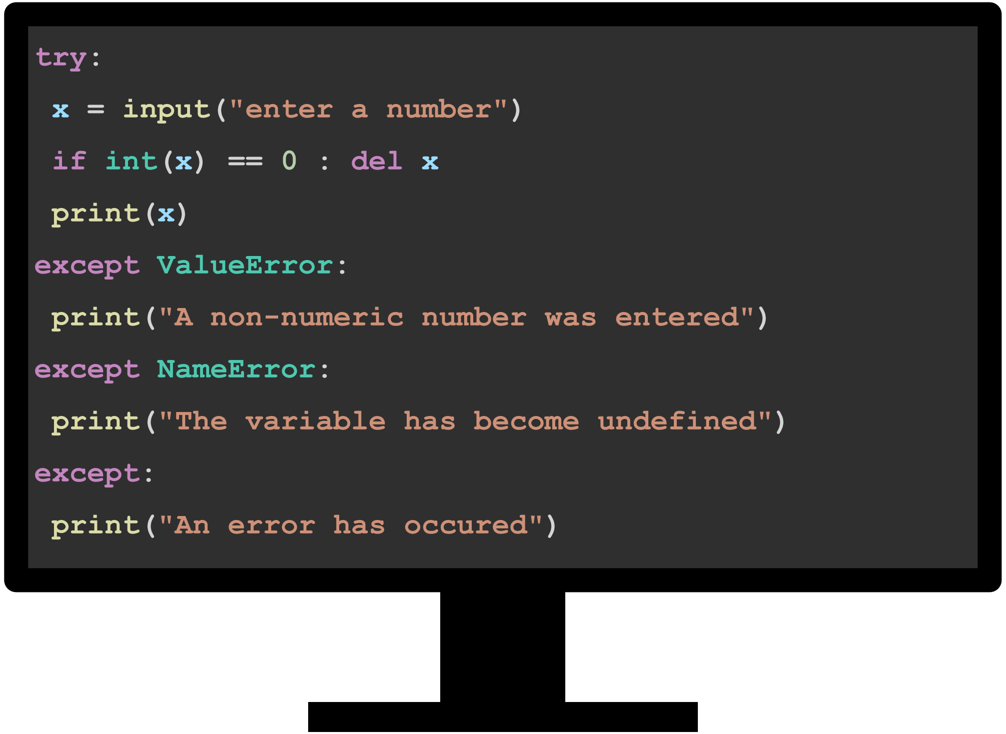
➔ If an error occurs within a **try block**, the entire block is skipped and the code within the **except block** is executed

```python
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

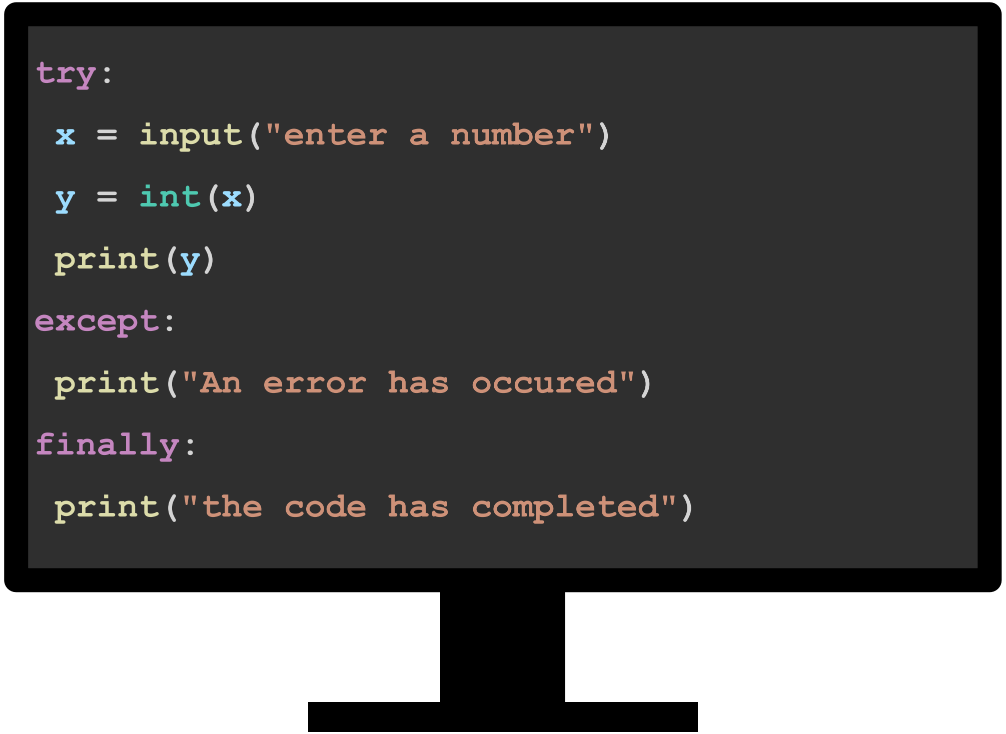
- ➔ Specific **errors** can be caught in their own except blocks
  - ◆ Each **error** can be handled differently
- ➔ An **except block** without a named error will catch any **error**
  - ◆ The general catch must be placed after any named **error**



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

Finally

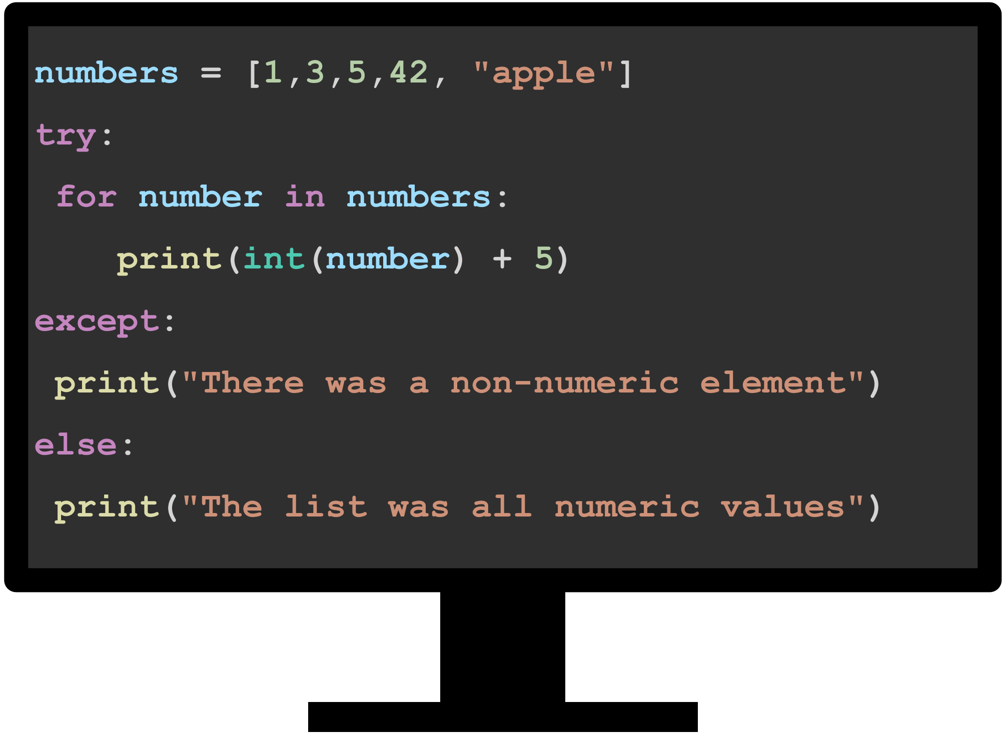
- ➔ A **finally** block will execute regardless of the status of the try/except blocks
- ➔ A **finally** block is a good place to perform any cleanup
    - ◆ Close connections
    - ◆ End timers
    - ◆ Cancel subscriptions



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

Else

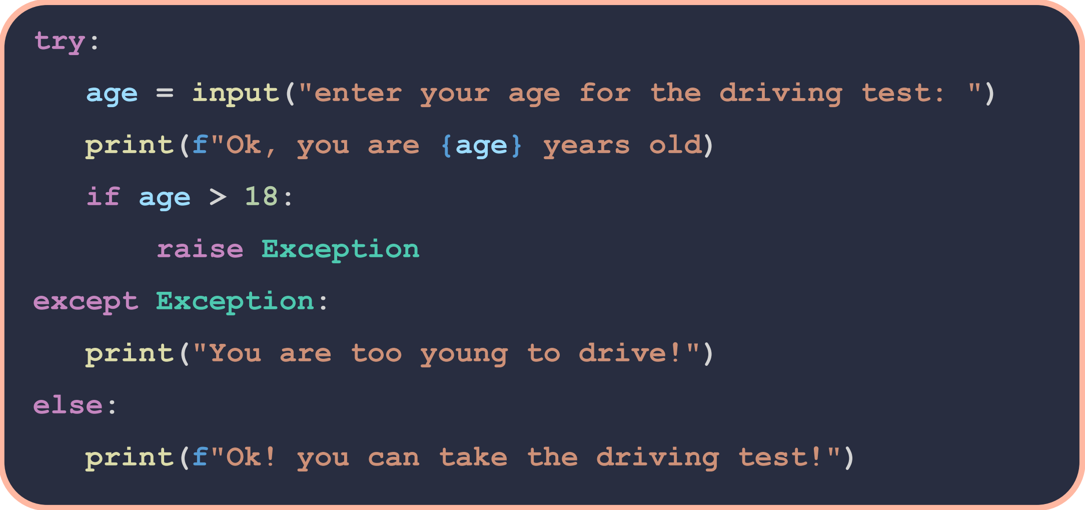
- ➔ An **else block** will only execute if the **try block** of the executed without **error**
- ➔ An **else block** is a good place to perform any actions that depend on the successful completion of the **try block**



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

Raise

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



```
try:
    age = input("enter your age for the driving test: ")
    print(f"Ok, you are {age} years old)
    if age > 18:
        raise Exception
except Exception:
    print("You are too young to drive!")
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
  - Optional: raise an Error if the employee is under 18
- Print each employee’s information in a formatted string

$$f(x)$$

Functions

Python Functions

- ➔ A Function is a block of code that will only execute when called
- ➔ In Python, functions are defined with the def keyword

```python
def say_hello():
    print("Hello World")
```

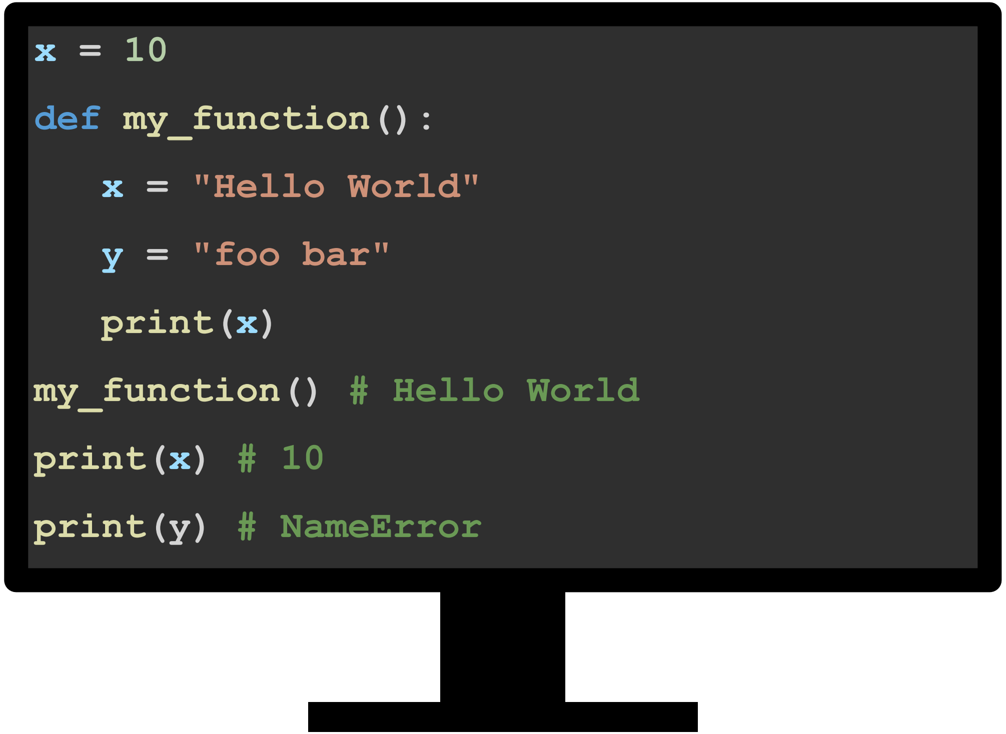
- ➔ Functions are called with parentheses

```python
say_hello() # Hello World
```

- ➔ Functions are Not hoisted in Python
  - ◆ Must be defined before they are called

Scope

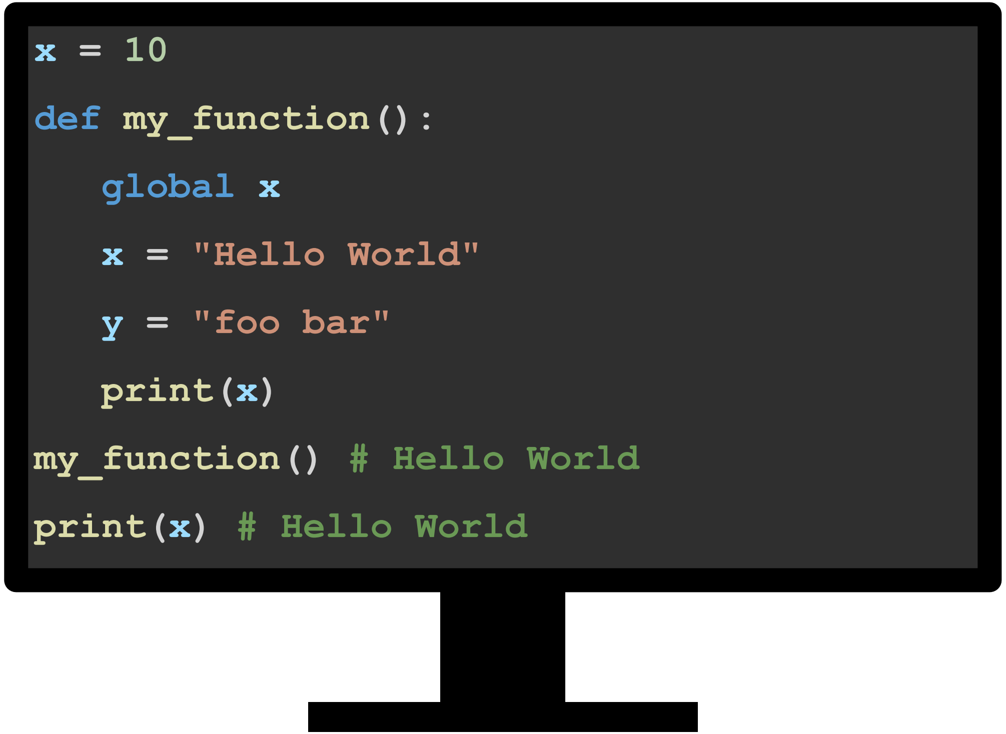
- ➔ **Variables** in Python are only visible in a **function** block
- ➔ Variables declared outside all functions are **global**
  - ◆ Can be accessed anywhere
- ➔ Variables declared inside a function are **local**
  - ◆ Can only be accessed in the function



```
x = 10
def my_function():
    x = "Hello World"
    y = "foo bar"
    print(x)
my_function() # Hello World
print(x) # 10
print(y) # NameError
```

Global Keyword

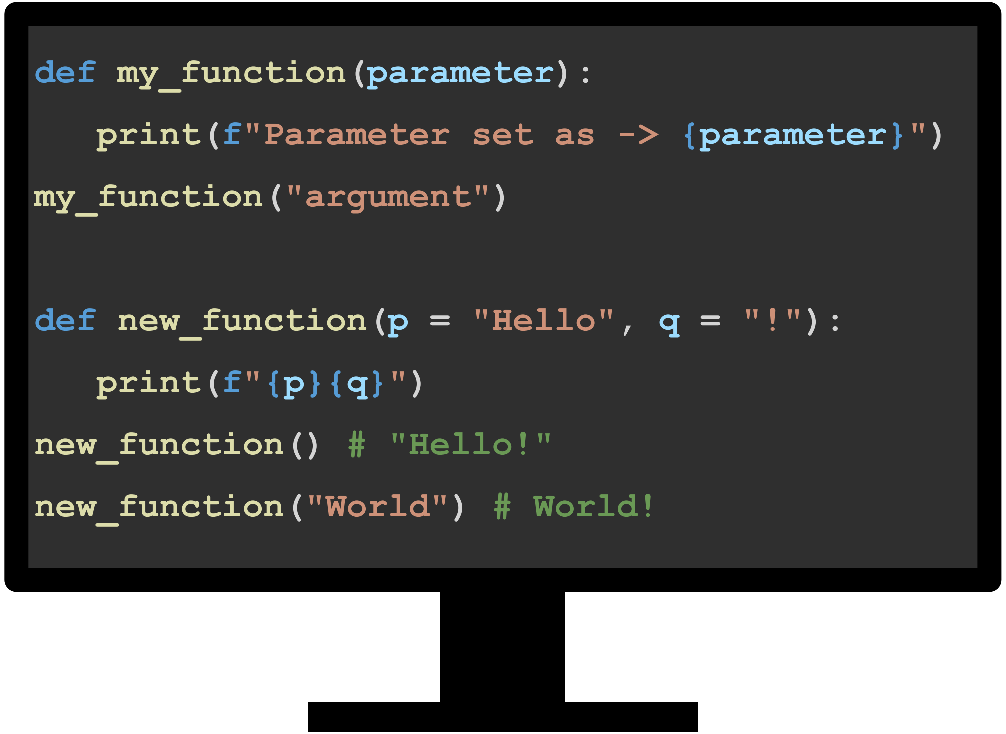
- ➔ Local variables and global variables with the same name are treated as two seperate variables
- ➔ Local variables can be made global using the **global** keyword



```
x = 10
def my_function():
   global x
   x = "Hello World"
   y = "foo bar"
   print(x)
my_function() # Hello World
print(x) # Hello World
```

Parameters

- ➔ A **function** can be passed a **parameter**
  - ◆ A named variable is created with function scope
  - ◆ No set types
  - ◆ The value passed to the function is called an **argument**
- ➔ A default value for a parameter can be set with **=**
  - ◆ The default value is set if no positional argument is passed



```
def my_function(parameter):
    print(f"Parameter set as -> {parameter}")
my_function("argument")

def new_function(p = "Hello", q = "!"):
    print(f"{p}{q}")
new_function()  # "Hello!"
new_function("World")  # World!
```

Keyword Arguments

➔ Python functions can set named parameters, and set those values as arguments

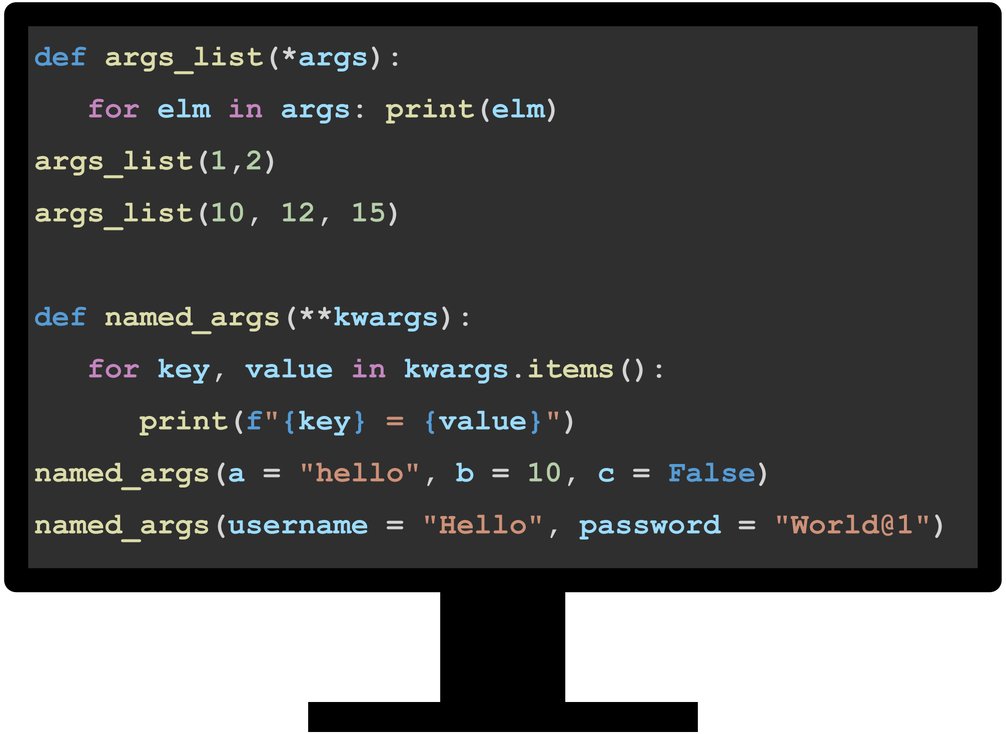
```python
def named_arguemnts(a,b,c):
   print(f"{a}, {b}, {c}")
named_arguemnts(a = "foo", b = "bar", c = "baz") # foo, bar, baz
```

➔ The order of arguments does not matter

```python
named_arguemnts(c = "first", a = "second", b = "third") # second, third, first
named_arguemnts(b = "first", c = "second", a = "third") # third, first, second
```

Argument Lists

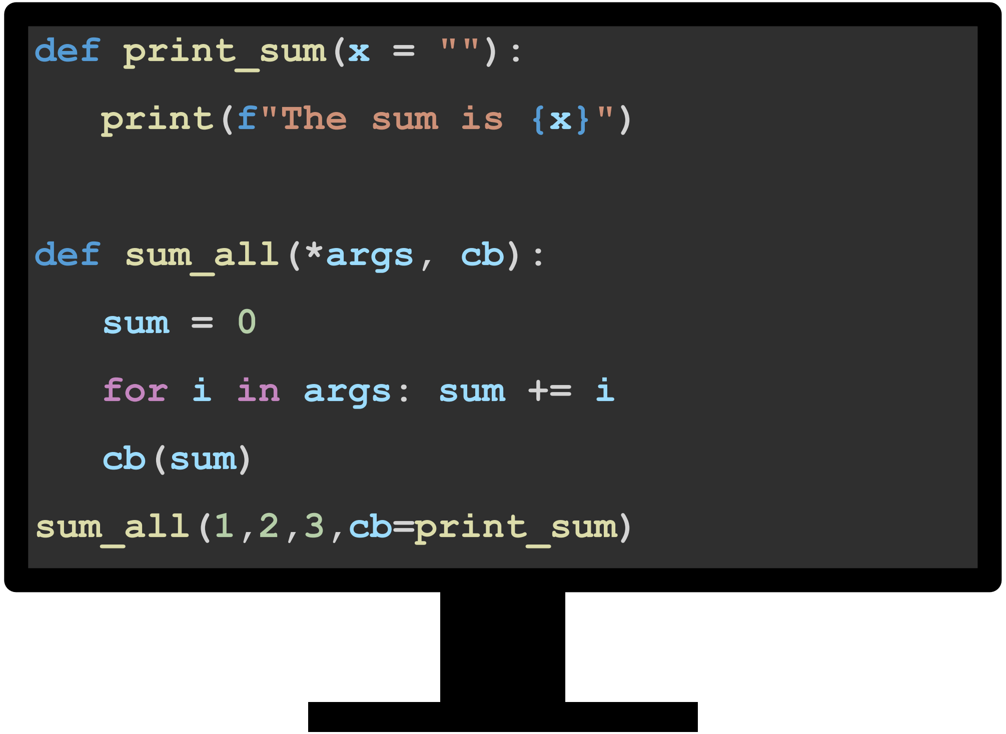
- ➔ *args
  - ◆ Common syntax
  - ◆ Function accepts any number of arguments as a **tuple**
- ➔ **kwargs
  - ◆ Common syntax
  - ◆ Function accepts any number of arguments as key/value pairs as a **dictionary**
  - ◆ = sets the pair



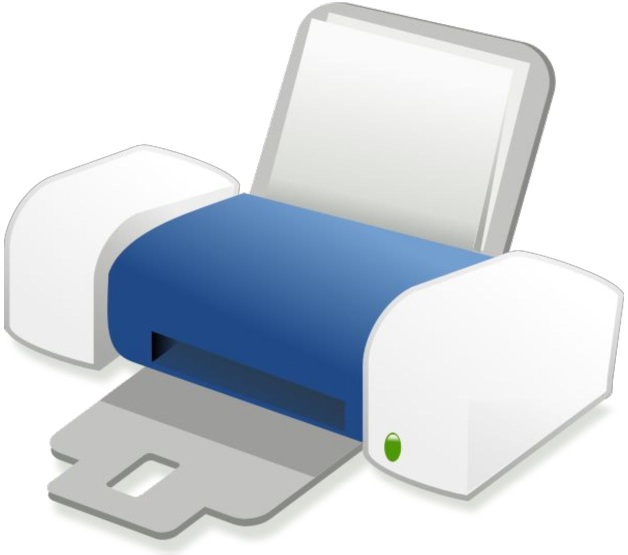
```python
def args_list(*args):
    for elm in args: print(elm)
args_list(1,2)
args_list(10, 12, 15)

def named_args(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} = {value}")
named_args(a = "hello", b = 10, c = False)
named_args(username = "Hello", password = "World@1")
```

Function Args

- ➔ Python functions can accept other functions as arguments
- ➔ Function arguments can be called as part of the outer function's execution



```
def print_sum(x = ""):
    print(f"The sum is {x}")

def sum_all(*args, cb):
    sum = 0
    for i in args: sum += i
    cb(sum)
sum_all(1,2,3,cb=print_sum)
```



The Print Function (cont.)

- ➔ The print function accepts an *args argument by default

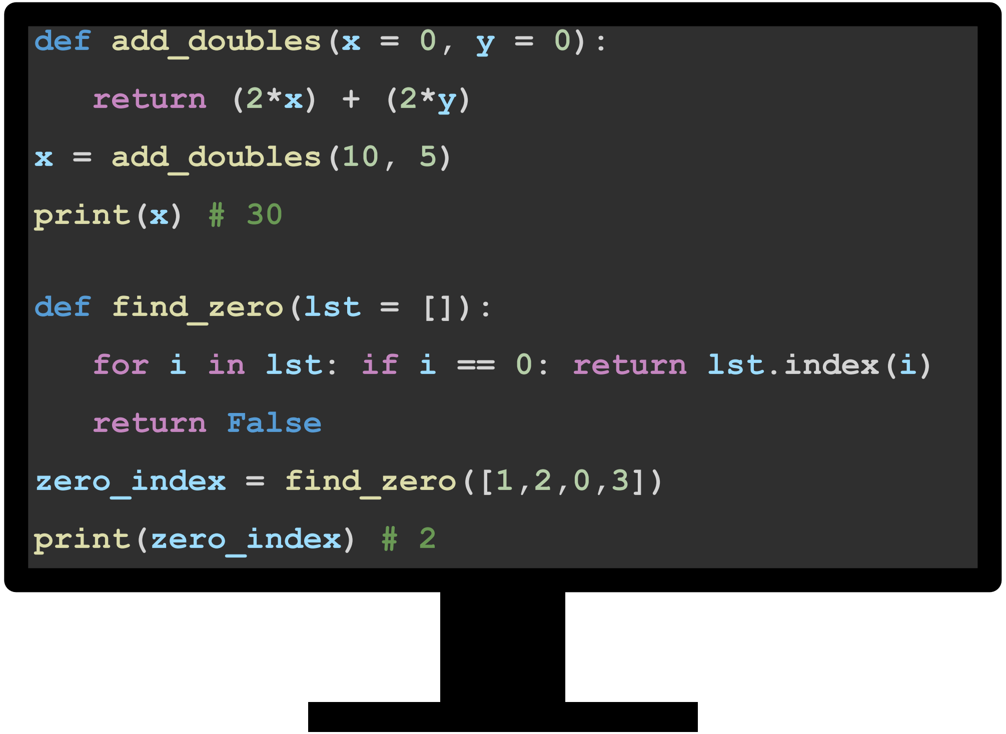
```
print ("Hello", "World", 42, False)
```

- ➔ print accepts named arguments
  - ◆ end = delimiter, defaults to newline (\n)
  - ◆ sep = separator between values. default space

```
print(1,2,3, sep="*", end="\t")
print(4,5,6, sep="@", end="\n\n")
```

Return

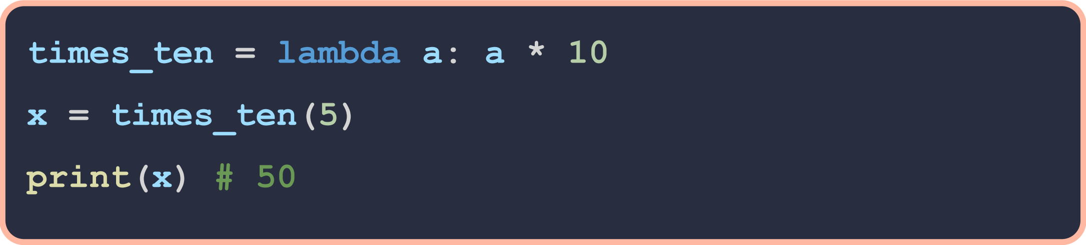
- ➔ The **return** keyword will assign the given value to the caller of the function
- ➔ A **return** statement will end a function
  - ◆ All local variables cleared from memory
  - ◆ All loops ended
- ➔ No defined **return** type



```
def add_doubles(x = 0, y = 0):
    return (2*x) + (2*y)
x = add_doubles(10, 5)
print(x) # 30

def find_zero(lst = []):
    for i in lst: if i == 0: return lst.index(i)
    return False
zero_index = find_zero([1,2,0,3])
print(zero_index) # 2
```

Lambda Functions

- Lambda Functions are small, anonymous functions
  - Use the **lambda** keyword
  - Must only be one line
  - Implicitly return value
  - Limited use is encouraged by the style guide
  - Are used in some core Python features
  - Used in Higher-Order Functions



```python
times_ten = lambda a: a * 10
x = times_ten(5)
print(x) # 50
```

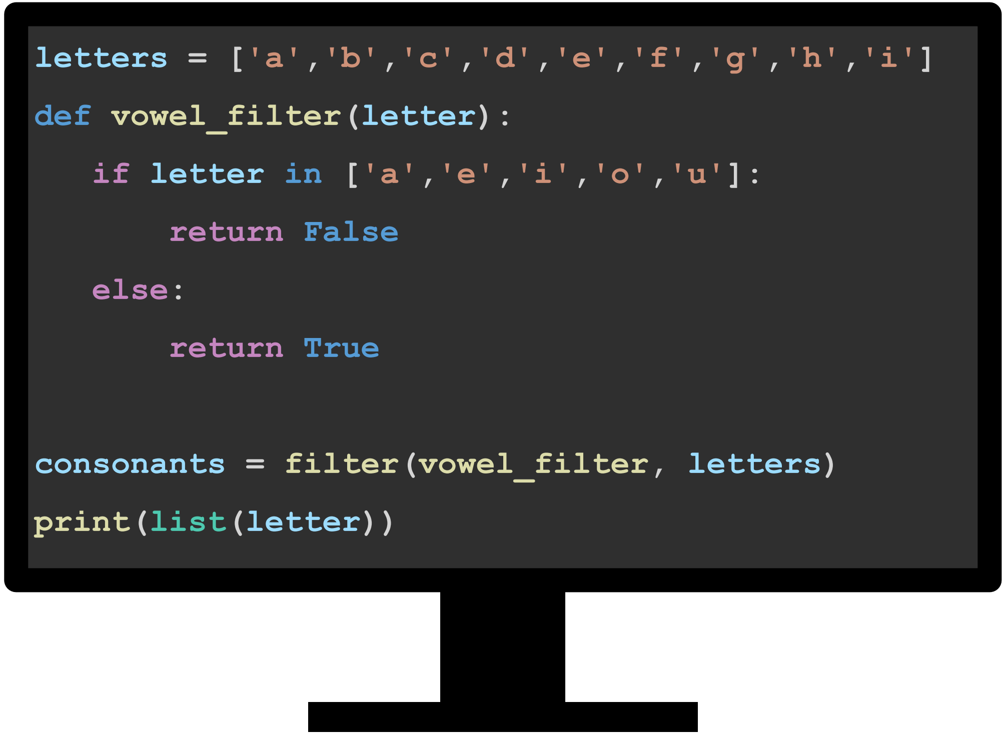
Functional Programming

Functional Programming

- **Functional Programming** is an alternative style to **imperative programming**
  - Both are mathematically equivalent
- Focuses on programming through pure, mathematical functions
  - **Immutable data, modularity, lazy evaluation, no side-effects, etc.**
- Functional Languages:
  - **Lisp, Haskell, F#**
- Imperative languages:
  - **C, Fortran, Python**
- Imperative languages can implement **functional style** features

Filter

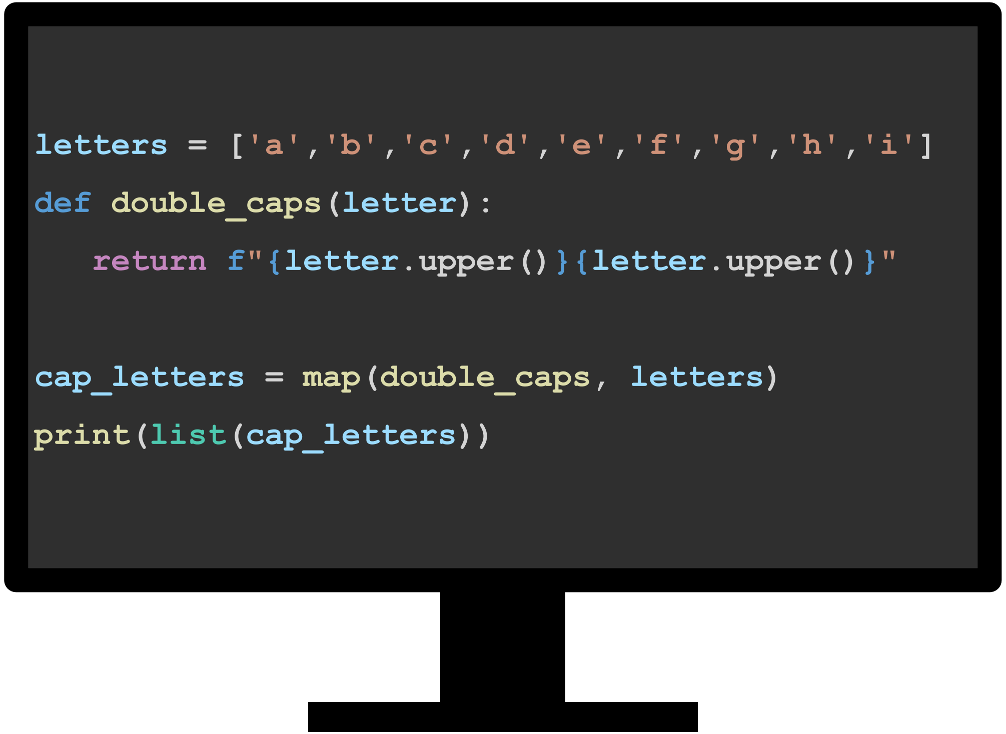
- ➔ Accepts a filtering function and a sequence
    - ◆ The filter returns a True or False for a single input
- ➔ Returns an iterator containing all elements that the filtering function returned True
- ➔ Often used with **lambdas**



```python
letters = ['a','b','c','d','e','f','g','h','i']
def vowel_filter(letter):
    if letter in ['a','e','i','o','u']:
        return False
    else:
        return True

consonants = filter(vowel_filter, letters)
print(list(letter))
```

Map

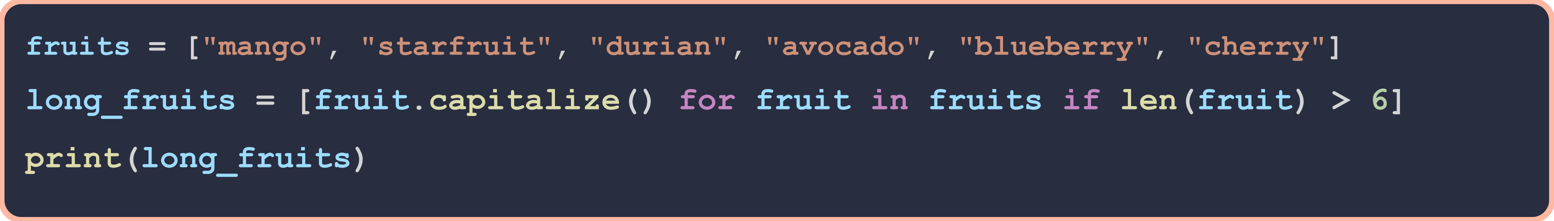
- ➔ Accepts a mapping function and a sequence
  - ◆ The map can return any single object
- ➔ Returns an iterator containing the return of the mapping function for each element of the sequence
- ➔ Often used with **lambdas**



```python
letters = ['a','b','c','d','e','f','g','h','i']
def double_caps(letter):
    return f"{letter.upper()}{letter.upper()}"

cap_letters = map(double_caps, letters)
print(list(cap_letters))
```

List Comprehension

- ➔ A **List Comprehension** (**listcomp**) is a compact way of processing a list
  - ◆ Returns a new list
  - ◆ [<**value or function return**> **for** <**variable**> **in** <**sequence**> (**optional condition**)]

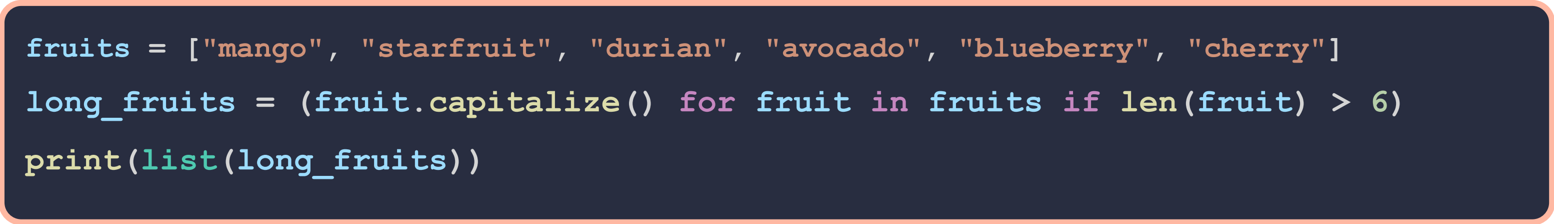


```python
fruits = ["mango", "starfruit", "durian", "avocado", "blueberry", "cherry"]
long_fruits = [fruit.capitalize() for fruit in fruits if len(fruit) > 6]
print(long_fruits)
```

- ➔ **List Comprehensions** are often preferred over Loops, Maps or Filters due to their compact syntax

Generator Expressions

- ➔ A **Generator Expression** is a compact way of processing a list
    - ◆ Returns an iterator



```
fruits = ["mango", "starfruit", "durian", "avocado", "blueberry", "cherry"]
long_fruits = (fruit.capitalize() for fruit in fruits if len(fruit) > 6)
print(list(long_fruits))
```

- ➔ **Generator Expressions** are preferred over **List Comprehensions** for very large data sets
    - ◆ Do no return a single data structure
    - ◆ Instead returns an iterator, which is lazily evaluated

FIN