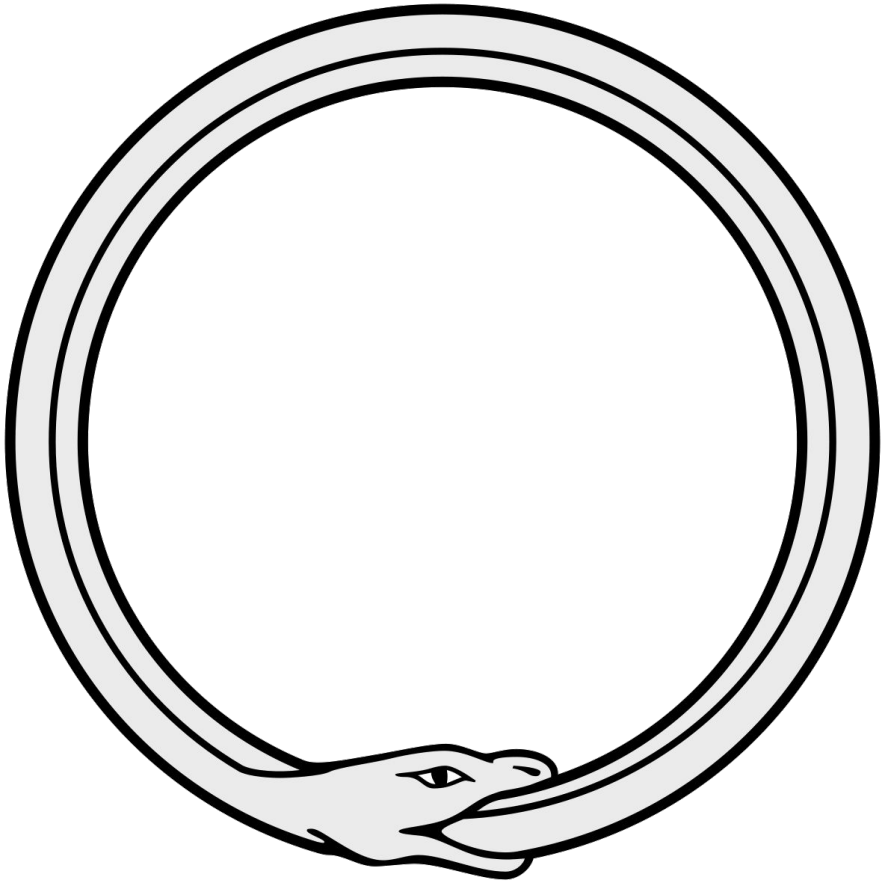
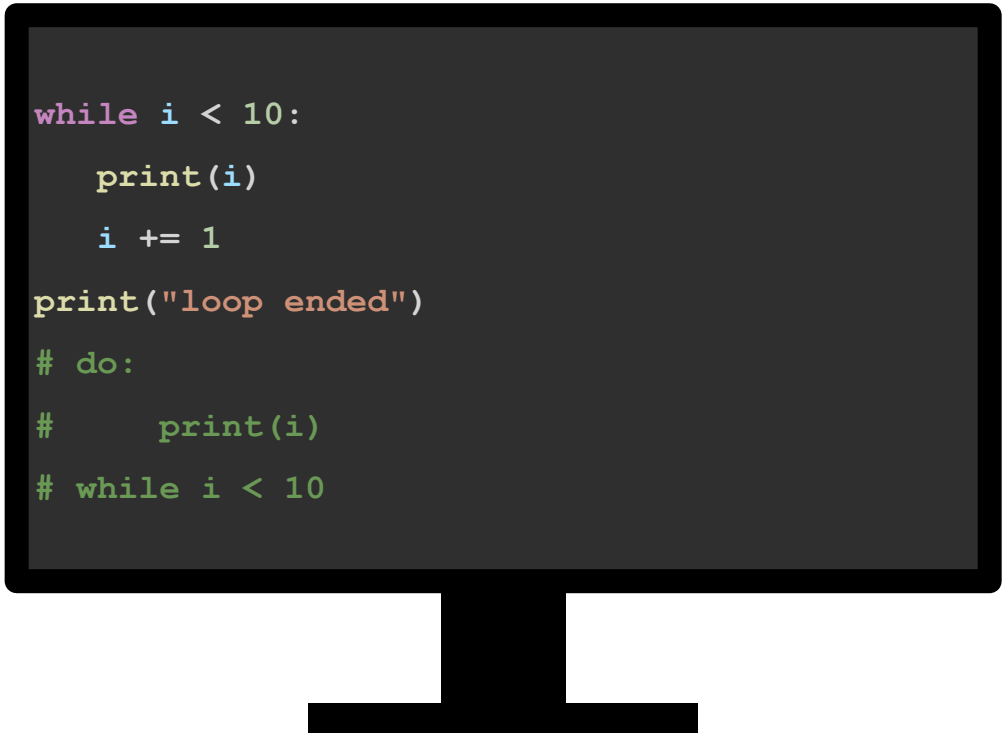


While Loops

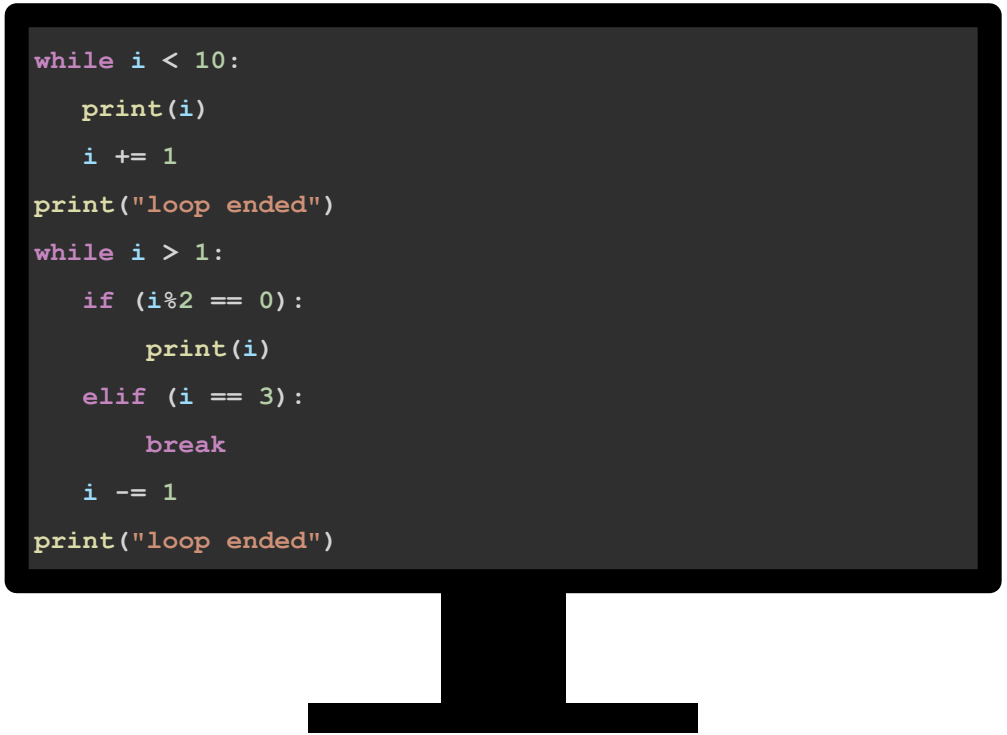
While Loop

- **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



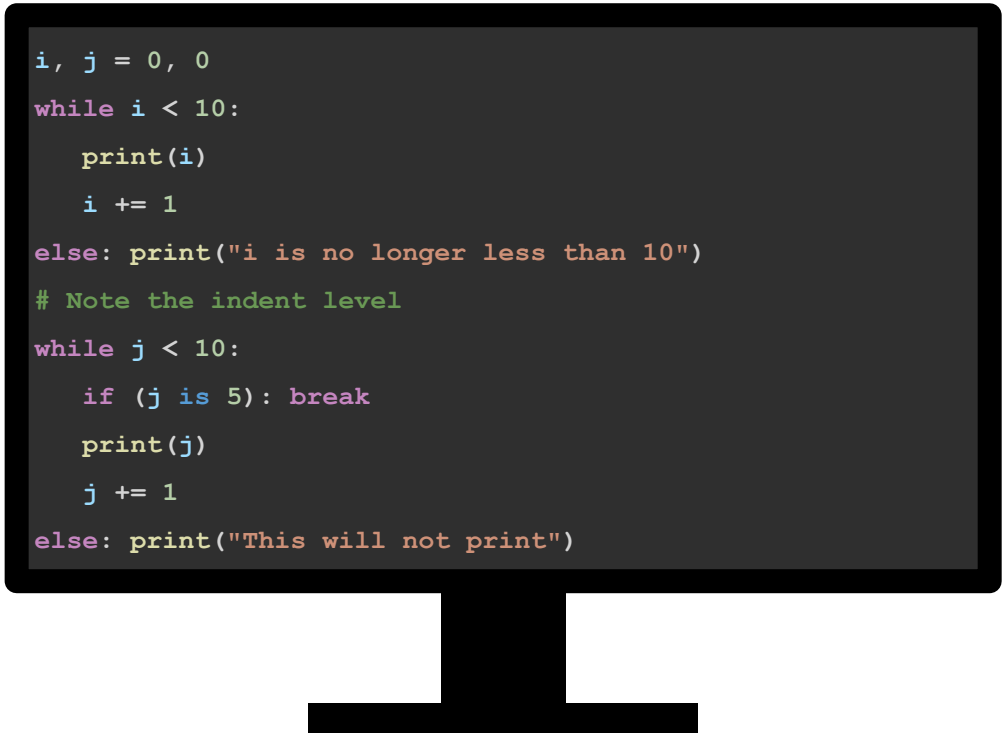
Break, Continue

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



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



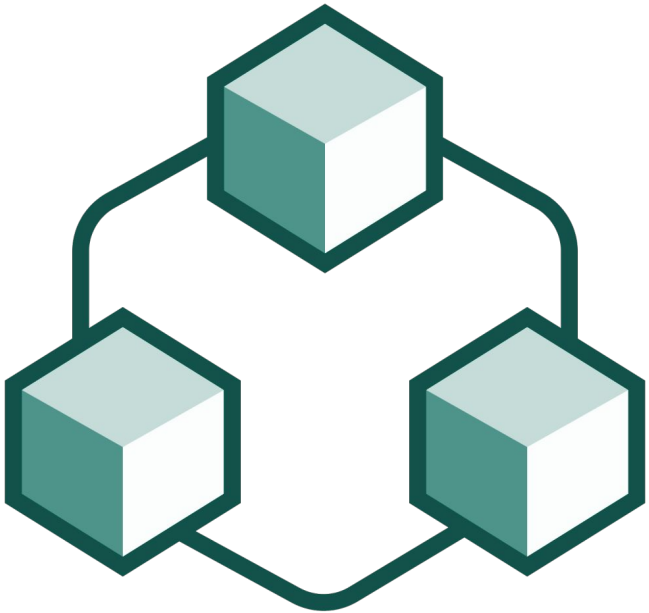
```
i, j = 0, 0
while i < 10:
   print(i)
   i += 1
else: print("i is no longer less than 10")
# Note the indent level
while j < 10:
   if (j is 5): break
   print(j)
   j += 1
else: print("This will not print")
```

Student Exercise

➔ Rock - Paper - Scissors

- Take your Rock-Paper-Scissors game and expand it!
- Prompt the user to ask them if they want to play again
- Repeat the game until the user decides to quit
- Keep track of the wins for players 1 and 2, and the draws
- **(OPTIONAL) - Give the option to play a game to best of 3 or best of 5**
- **(OPTIONAL) - Play against the computer**

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

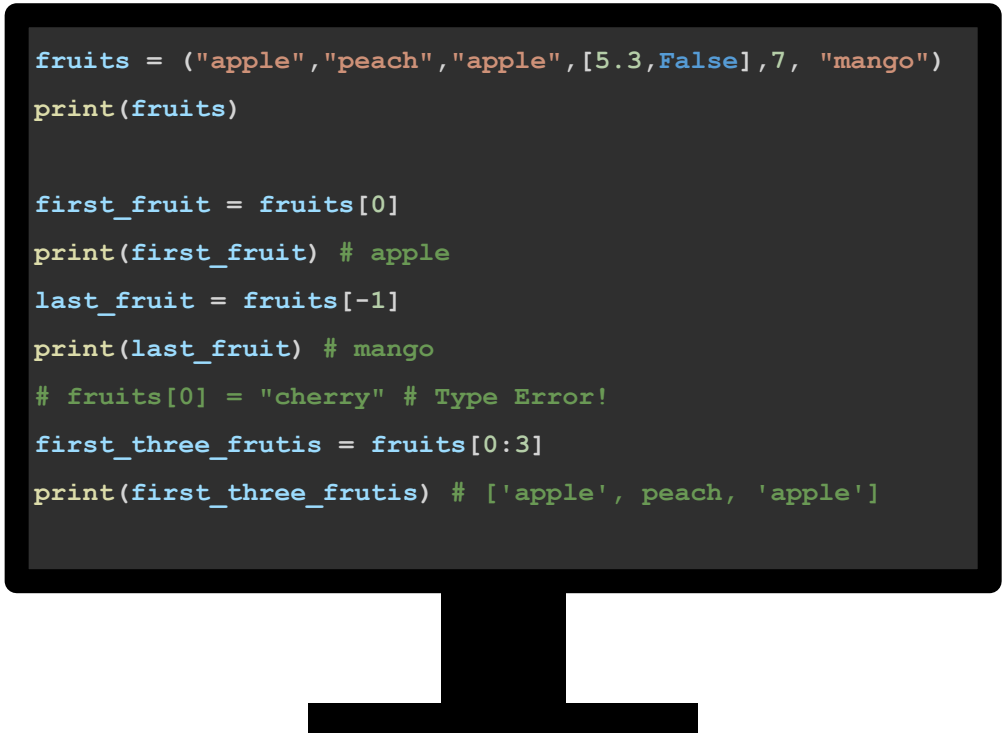
```
fruits = ["apple","peach","apple",[5.3,False],7, "mango"]
print(fruits)

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 Collection Methods

Method	Description
.add(element)	Adds an element to the end of the set
.clear()	Removes all elements from the set
.copy()	Returns a copy of the set
.discard(element)	Removes the element from the set
.pop()	Removes a random element and returns it
.remove(element)	Removes the given element and returns it
.update(set)	Updates the set with a union of the given set

Set Math Methods

Method	Description
.difference(set)	Returns a set containing the difference between sets
.difference_update(set)	Removes the items in this set that are also included in another, specified set
.intersection(set)	Returns a set that is the intersection of the sets
.intersection_update(set)	Removes the items in this set that are not present in other, specified set(s)
.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
.symmetric_difference()	Returns a set with the symmetric differences of two sets
.symmetric_difference_update()	inserts the symmetric differences from this set and another
.union(set)	Returns the union of the two sets

Dictionary

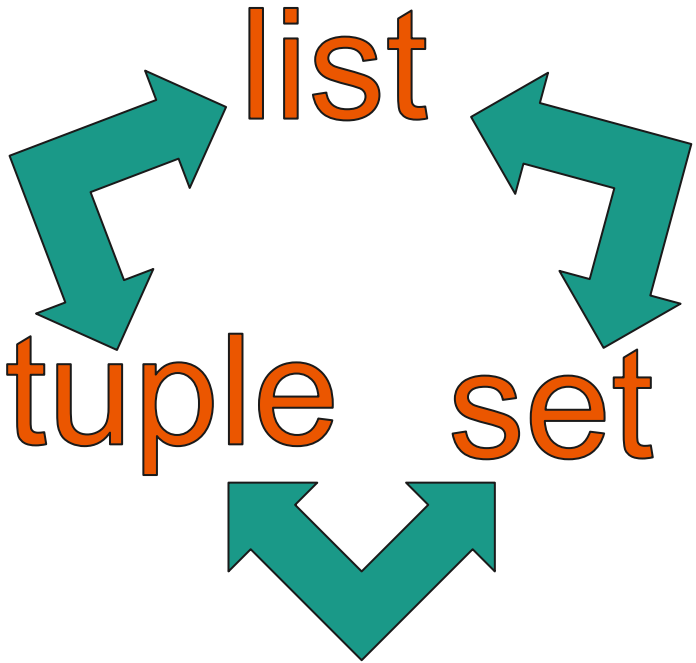
- Declared with curly brackets { }
  - Keys must be strings or numbers
- Elements can be accessed by their keys with the square brackets [ ]
  - Can be iterated over to access elements
  - Duplicate keys update value

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

Casting Between Collections

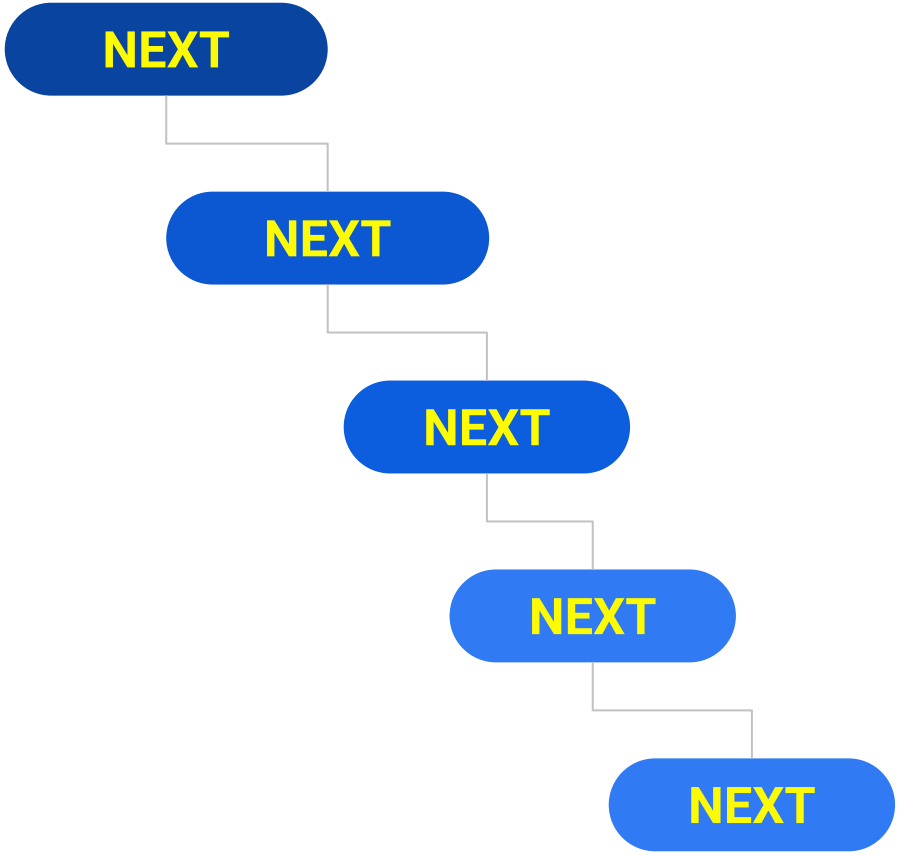
- Python can easily switch between one data structure an another
- **set()**
  - can turn a list or tuple into a set
- **tuple()**
  - can turn a set or list into a tuple
- **list()**
  - can turn a set or tuple into a list
- Casting from a tuple to a list can allow for altering elements, then casting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



Student Exercise

- Create a **dictionary** and prompt the user for their "name", "age", and "years coding" as 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 values to the dictionary you created, with appropriate keys
  - format a print statement to print the relevant data to the console

Iterators

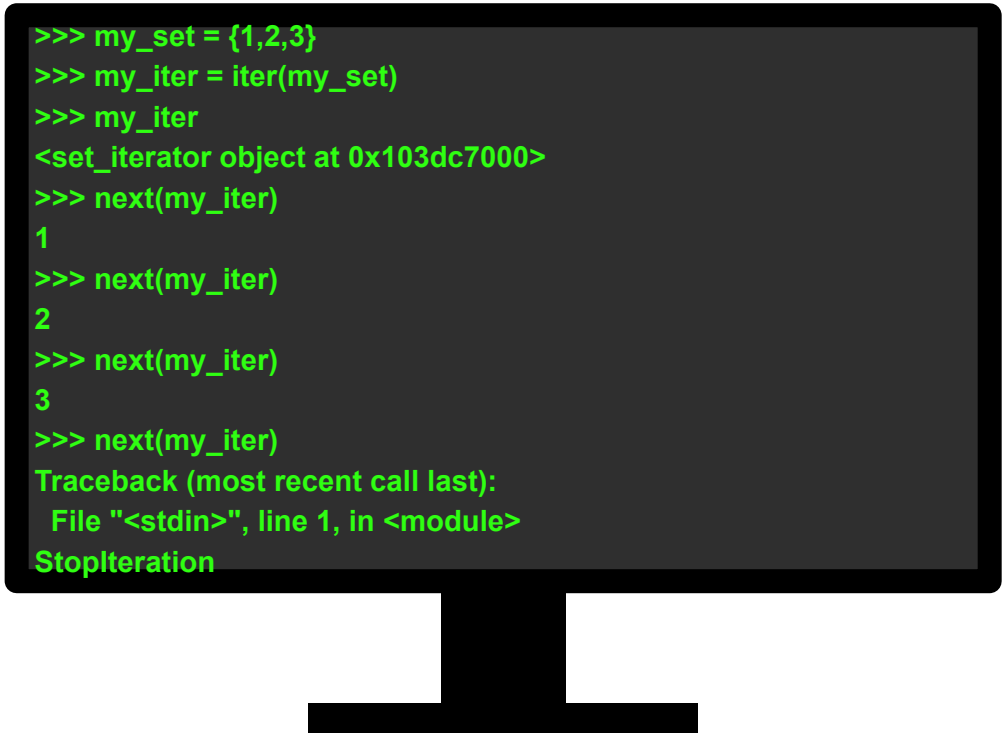


Iterators

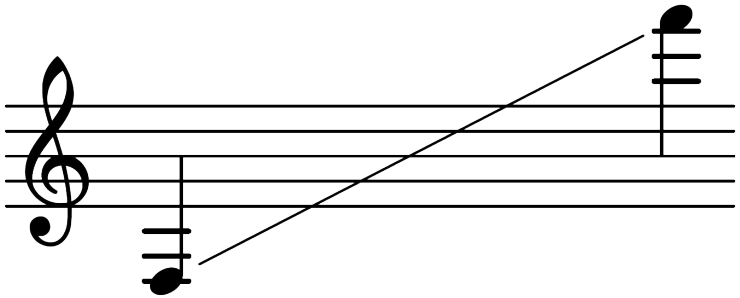
- In Python, an iterable is any object that contains a countable number of items
  - **Lists**, **Dictionaries**, **Tuples**, and **Sets** are all iterable objects
  - **Strings**, as character lists, are also iterable
- An iterator is an object that contains the following methods:
  - **__iter__()**: 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")**

Iterator

- An iterator can be created from calling the global **iter()** function on an iterable object
- Calling **next()** advances the iterator
- When there is no more data, a **StopIteration** exception is raised
- Custom Iterators can be created



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)
```

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

For Loops

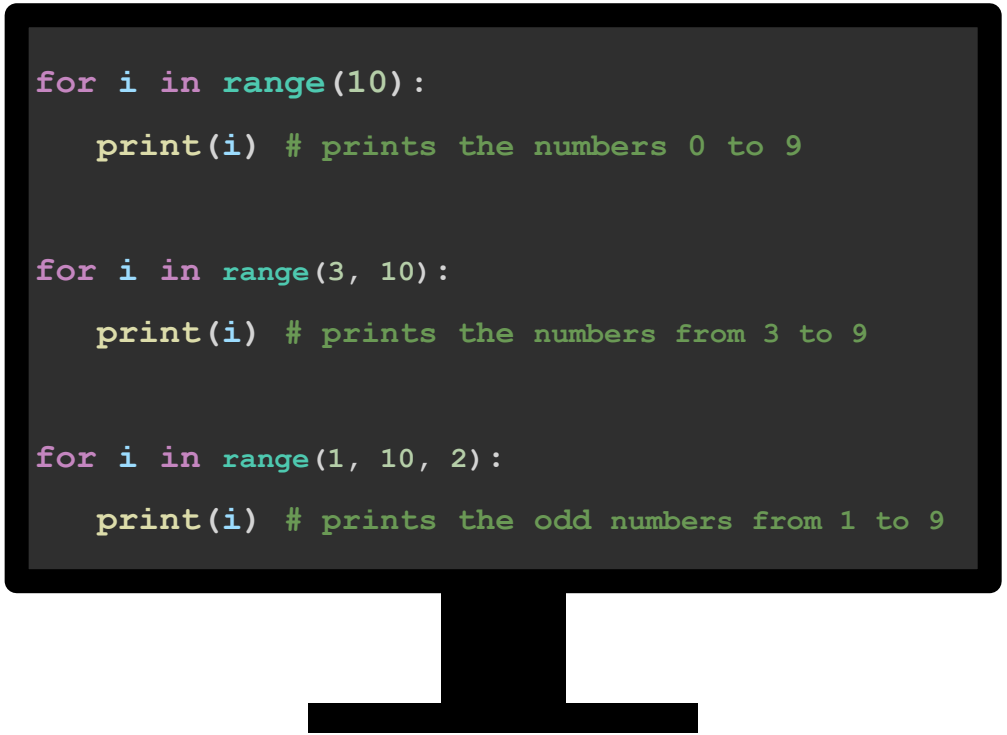
For Construct

- **For Loops** in Python will execute over any sequence
  - 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

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

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

For … range( )

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



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 to the list
  - 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**