

# SEAS 6414 – Python Applications in Data Analytics

---

**THE GEORGE  
WASHINGTON  
UNIVERSITY**

---

WASHINGTON, DC

Dr. Adewale Akinfaderin

Lecture 2 - Spring 2024

# For Loop

- 1 **For Loop Syntax:** For loops are used for iterating over collections (like lists or tuples) or iterators, following the standard syntax `for item in collection:`.
- 2 **Continue Keyword:** The `continue` keyword advances the loop to the next iteration, skipping the current loop block's remainder.
- 3 **Break Keyword:** The `break` keyword exits the innermost for loop entirely, while any outer for loops continue to run.
- 4 **Unpacking in For Loops:** If elements in the collection or iterator are sequences (such as tuples or lists), they can be conveniently unpacked into variables directly in the for loop statement.

# For Loop

```
[1]: sequence = [1, 2, None, 4, None, 5]
    total = 0
    for value in sequence:
        if value is None:
            continue
        total += value
```

```
[2]: total
```

```
[2]: 12
```

```
[3]: sequence = [1, 2, 0, 4, 6, 5, 2, 1]
    total_until_5 = 0
    for value in sequence:
        if value == 5:
            break
        total_until_5 += value
```

```
[4]: total_until_5
```

```
[4]: 13
```

# For Loop

```
[6]: for i in range(4):  
      for j in range(4):  
          if j > i:  
              break  
          print((i, j))
```

```
(0, 0)  
(1, 0)  
(1, 1)  
(2, 0)  
(2, 1)  
(2, 2)  
(3, 0)  
(3, 1)  
(3, 2)  
(3, 3)
```

# While Loop

- ① **While Loop Condition:** A while loop repeatedly executes a block of code as long as a specified condition evaluates to True.
- ② **Terminating a While Loop:** The loop can be explicitly terminated using the `break` keyword, even if the condition has not yet evaluated to False.

```
[7]: x = 256
    total = 0
    while x > 0:
        if total > 500:
            break
        total += x
        x = x // 2
```

```
[9]: total
```

```
[9]: 504
```

# Pass

- 1 **Pass as a No-Op Statement:** The 'pass' statement in Python is effectively a "no-op" (no operation) statement, used in scenarios where a statement is syntactically required but no action is intended.
- 2 **Usage in Blocks:** It serves as a placeholder in code blocks where no action is needed or for not-yet-implemented code, necessary due to Python's whitespace-based block delimitation.

```
[12]: x = -7

if x < 0:
    print("negative!")
elif x == 0:
    print("it's Zero!")
    pass
else:
    print("positive!")

negative!
```

# Range

- 1 **Range Function Basics:** The 'range' function in Python generates a sequence of evenly spaced integers, commonly used for creating integer sequences.
- 2 **Parameters of Range:** It accepts start, end, and step arguments, where the step value can be negative. The function generates integers up to, but not including, the endpoint.
- 3 **Iterating with Range:** 'range' is often used for iterating through sequences by index, such as in for loops.
- 4 **Efficiency and Memory Usage:** While 'range' can generate large sequences, it is memory-efficient, as it does not store all the integers at once but creates them on-the-fly.

# Range

```
[13]: range(10)
      list(range(10))
```

```
[13]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[15]: list(range(0, 20, 2))
```

```
[15]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
[16]: list(range(5, 0, -1))
```

```
[16]: [5, 4, 3, 2, 1]
```

```
[17]: seq = [1, 2, 3, 4]
      for i in range(len(seq)):
          print(f"element {i}: {seq[i]}")
```

```
element 0: 1
element 1: 2
element 2: 3
element 3: 4
```

```
[18]: total = 0
      for i in range(100_000):
          # % is the modulo operator
          if i % 3 == 0 or i % 5 == 0:
              total += i
      print(total)
```

```
2333316668
```



# Tuples

- 1 **Importance of Data Structures:** Python's data structures, like tuples, lists, and dictionaries, are fundamental and mastering them is crucial for becoming a proficient Python programmer.
- 2 **Tuple Characteristics:** A tuple is an immutable sequence of Python objects with fixed length. Once created, its elements cannot be changed. Tuples are typically defined with values enclosed in parentheses.
- 3 **Tuple Creation and Conversion:** Tuples can be created without parentheses in some contexts, and any sequence or iterator can be converted to a tuple using the 'tuple' function.
- 4 **Accessing Tuple Elements:** Elements in a tuple can be accessed using square brackets '[]', similar to other sequence types, with Python using 0-based indexing.

# Tuples

```
[19]: tup = (4, 5, 6)  
tup
```

```
[19]: (4, 5, 6)
```

```
[20]: tup = 4, 5, 6  
tup
```

```
[20]: (4, 5, 6)
```

```
[21]: tuple([4, 0, 2])  
tup = tuple('string')  
tup
```

```
[21]: ('s', 't', 'r', 'i', 'n', 'g')
```

```
[22]: tup[0]
```

```
[22]: 's'
```

# Tuples

- 1 **Defining Tuples in Expressions:** In complex expressions, enclosing tuple values in parentheses is often necessary, such as when creating a tuple of tuples.
- 2 **Tuple Immutability:** Though objects within a tuple may be mutable, the tuple itself is immutable, meaning the objects stored in each slot cannot be modified post-creation.
- 3 **Mutable Elements in Tuples:** If a tuple contains mutable objects (e.g., lists), these objects can be modified in place.
- 4 **Concatenating Tuples:** Tuples can be combined using the '+' operator, producing longer tuples.
- 5 **Multiplying Tuples:** Similar to lists, multiplying a tuple by an integer results in concatenating that many copies of the tuple.

# Tuples

```
[24]: nested_tup = (4, 5, 6), (7, 8)
      nested_tup
```

```
[24]: ((4, 5, 6), (7, 8))
```

```
[25]: nested_tup[0]
```

```
[25]: (4, 5, 6)
```

```
[26]: nested_tup[1]
```

```
[26]: (7, 8)
```

```
[27]: tup = tuple(['foo', [1, 2], True])
      tup[2] = False
```

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[27], line 2
      1 tup = tuple(['foo', [1, 2], True])
----> 2 tup[2] = False

TypeError: 'tuple' object does not support item assignment
```

```
[28]: tup[1].append(3)
      tup
```

```
[28]: ('foo', [1, 2, 3], True)
```

# Tuples

- 1 **Tuple Unpacking:** Python attempts to unpack values on the right-hand side of the equals sign when assigning to a tuple-like expression of variables.
- 2 **Nested Tuple Unpacking:** Python supports unpacking even for sequences with nested tuples, allowing for complex data structures to be easily deconstructed.
- 3 **Variable Swapping:** Tuple unpacking facilitates easy variable swapping, a process that is more cumbersome in many other programming languages.
- 4 **Unpacking in Iteration:** Variable unpacking is commonly used for iterating over sequences of tuples or lists

# Tuples

```
[31]: tup[1].append(3)  
tup
```

```
[31]: ('foo', [1, 2, 3], True)
```

```
[32]: (4, None, 'foo') + (6, 0) + ('bar',)
```

```
[32]: (4, None, 'foo', 6, 0, 'bar')
```

```
[33]: ('foo', 'bar') * 4
```

```
[33]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

# Tuples

```
[34]: tup = (4, 5, 6)
      a, b, c = tup
      b
```

```
[34]: 5
```

```
[35]: tup = 4, 5, (6, 7)
      a, b, (c, d) = tup
      d
```

```
[35]: 7
```

```
[36]: a, b = 1, 2
      a
      b
      b, a = a, b
      a
      b
```

```
[36]: 1
```

```
[37]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
      for a, b, c in seq:
          print(f'a={a}, b={b}, c={c}')

a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

# Tuple Methods

- 1 **Limited Tuple Methods:** Due to their immutable nature, tuples have very few instance methods, reflecting their inability to modify size or contents.
- 2 **Count Method:** One useful method that tuples share with lists is 'count', which tallies the number of times a specific value occurs within the tuple.

```
[40]: a = (1, 2, 2, 2, 3, 4, 2)
      a.count(2)
```

```
[40]: 4
```



# Lists

- 1 **Lists are Mutable:** Unlike tuples, lists in Python are variable in length and their contents can be modified in place, which makes them mutable.
- 2 **Defining Lists:** Lists can be defined using square brackets '[]' or by using the 'list' type function.
- 3 **Semantic Similarity to Tuples:** Lists and tuples are similar in function and can often be used interchangeably, with the main difference being that tuples are immutable.
- 4 **List Function for Data Processing:** The 'list' function is widely used in data processing to materialize iterators or generator expressions, making it a crucial tool for handling data sequences.

# Lists

```
[41]: a_list = [2, 3, 7, None]

      tup = ("foo", "bar", "baz")
      b_list = list(tup)
      b_list
      b_list[1] = "peekaboo"
      b_list
```

```
[41]: ['foo', 'peekaboo', 'baz']
```

```
[42]: gen = range(10)
      gen
      list(gen)
```

```
[42]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Adding and Removing Elements

Elements can be appended to the end of the list with the `append` method:

```
In [51]: b_list.append("dwarf")  
  
In [52]: b_list  
Out[52]: ['foo', 'peekaboo', 'baz', 'dwarf']
```



Using `insert` you can insert an element at a specific location in the list:

```
In [53]: b_list.insert(1, "red")  
  
In [54]: b_list  
Out[54]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```



The insertion index must be between 0 and the length of the list, inclusive.

# Adding and Removing Elements

The inverse operation to `insert` is `pop`, which removes and returns an element at a particular index:

```
In [55]: b_list.pop(2)
Out[55]: 'peekaboo'
```

```
In [56]: b_list
Out[56]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value with `remove`, which locates the first such value and removes it from the list:

```
In [57]: b_list.append("foo")

In [58]: b_list
Out[58]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [59]: b_list.remove("foo")

In [60]: b_list
Out[60]: ['red', 'baz', 'dwarf', 'foo']
```

# Adding and Removing Elements

If performance is not a concern, by using `append` and `remove`, you can use a Python list as a set-like data structure (although Python has actual set objects, discussed later).

Check if a list contains a value using the `in` keyword:

```
In [61]: "dwarf" in b_list  
Out[61]: True
```



The keyword `not` can be used to negate `in`:

```
In [62]: "dwarf" not in b_list  
Out[62]: False
```



Checking whether a list contains a value is a lot slower than doing so with dictionaries and sets (to be introduced shortly), as Python makes a linear scan across the values of the list, whereas it can check the others (based on hash tables) in constant time.

# Concatenating and Combining Lists

Similar to tuples, adding two lists together with `+` concatenates them:

```
In [63]: [4, None, "foo"] + [7, 8, (2, 3)]  
Out[63]: [4, None, 'foo', 7, 8, (2, 3)]
```



If you have a list already defined, you can append multiple elements to it using the `extend` method:

```
In [64]: x = [4, None, "foo"]  
  
In [65]: x.extend([7, 8, (2, 3)])  
  
In [66]: x  
Out[66]: [4, None, 'foo', 7, 8, (2, 3)]
```



# Concatenating and Combining Lists

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus:

```
everything = []  
for chunk in list_of_lists:  
    everything.extend(chunk)
```



is faster than the concatenative alternative:

```
everything = []  
for chunk in list_of_lists:  
    everything = everything + chunk
```



# Sorting

You can sort a list in place (without creating a new object) by calling its `sort` function:

```
In [67]: a = [7, 2, 5, 1, 3]
```

```
In [68]: a.sort()
```

```
In [69]: a
```

```
Out[69]: [1, 2, 3, 5, 7]
```



`sort` has a few options that will occasionally come in handy. One is the ability to pass a secondary *sort* key—that is, a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [70]: b = ["saw", "small", "He", "foxes", "six"]
```

```
In [71]: b.sort(key=len)
```

```
In [72]: b
```

```
Out[72]: ['He', 'saw', 'six', 'small', 'foxes']
```



Soon, we'll look at the `sorted` function, which can produce a sorted copy of a general sequence.





# Slicing

You can select sections of most sequence types by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
```

```
In [74]: seq[1:5]
```

```
Out[74]: [2, 3, 7, 5]
```

Slices can also be assigned with a sequence:

```
In [75]: seq[3:5] = [6, 3]
```

```
In [76]: seq
```

```
Out[76]: [7, 2, 3, 6, 3, 6, 0, 1]
```

While the element at the `start` index is included, the `stop` index is *not included*, so that the number of elements in the result is `stop - start`.

# Slicing

Either the `start` or `stop` can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively:

```
In [77]: seq[:5]  
Out[77]: [7, 2, 3, 6, 3]
```

```
In [78]: seq[3:]  
Out[78]: [6, 3, 6, 0, 1]
```

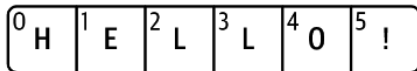
Negative indices slice the sequence relative to the end:

```
In [79]: seq[-4:]  
Out[79]: [3, 6, 0, 1]
```

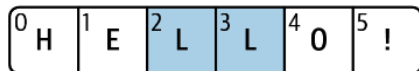
```
In [80]: seq[-6:-2]  
Out[80]: [3, 6, 3, 6]
```

# Slicing

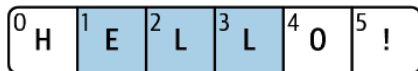
Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See [Figure 3.1](#) for a helpful illustration of slicing with positive and negative integers. In the figure, the indices are shown at the "bin edges" to help show where the slice selections start and stop using positive or negative indices.



0    1    2    3    4    5    6  
-6   -5   -4   -3   -2   -1



string[2:4]



string[-5:-2]

# Slicing

A `step` can also be used after a second colon to, say, take every other element:

```
In [81]: seq[::2]  
Out[81]: [7, 3, 3, 0]
```



A clever use of this is to pass `-1`, which has the useful effect of reversing a list or tuple:

```
In [82]: seq[::-1]  
Out[82]: [1, 0, 6, 3, 6, 3, 2, 7]
```



-- --

# Dictionary

The dictionary or `dict` may be the most important built-in Python data structure. In other programming languages, dictionaries are sometimes called *hash maps* or *associative arrays*. A dictionary stores a collection of *key-value* pairs, where *key* and *value* are Python objects. Each key is associated with a value so that a value can be conveniently retrieved, inserted, modified, or deleted given a particular key. One approach for creating a dictionary is to use curly braces `{}` and colons to separate keys and values:

```
In [83]: empty_dict = {}
```


```
In [84]: d1 = {"a": "some value", "b": [1, 2, 3, 4]}
```

```
In [85]: d1
```

```
Out[85]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

# Dictionary

You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

```
In [86]: d1[7] = "an integer" 
```

```
In [87]: d1
```

```
Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

```
In [88]: d1["b"]
```

```
Out[88]: [1, 2, 3, 4]
```

You can check if a dictionary contains a key using the same syntax used for checking whether a list or tuple contains a value:

```
In [89]: "b" in d1 
```

```
Out[89]: True
```

# Dictionary

You can delete values using either the `del` keyword or the `pop` method (which simultaneously returns the value and deletes the key):

```
In [90]: d1[5] = "some value"
```

```
In [91]: d1
```

```
Out[91]:
```

```
{'a': 'some value',  
 'b': [1, 2, 3, 4],  
  7: 'an integer',  
  5: 'some value'}
```

```
In [92]: d1["dummy"] = "another value"
```

```
In [93]: d1
```

```
Out[93]:
```

```
{'a': 'some value',  
 'b': [1, 2, 3, 4],  
  7: 'an integer',  
  5: 'some value',  
 'dummy': 'another value'}
```

```
In [94]: del d1[5]
```

```
In [95]: d1
```

```
Out[95]:
```

```
{'a': 'some value',  
 'b': [1, 2, 3, 4],  
  7: 'an integer',  
 'dummy': 'another value'}
```

# Dictionary

The `keys` and `values` method gives you iterators of the dictionary's keys and values, respectively. The order of the keys depends on the order of their insertion, and these functions output the keys and values in the same respective order:

```
In [99]: list(d1.keys())  
Out[99]: ['a', 'b', 7]  
  
In [100]: list(d1.values())  
Out[100]: ['some value', [1, 2, 3, 4], 'an integer']
```

If you need to iterate over both the keys and values, you can use the `items` method to iterate over the keys and values as 2-tuples:

```
In [101]: list(d1.items())  
Out[101]: [('a', 'some value'), ('b', [1, 2, 3, 4]), (7, 'an integer')]
```

You can merge one dictionary into another using the `update` method:

```
In [102]: d1.update({"b": "foo", "c": 12})  
  
In [103]: d1  
Out[103]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

The `update` method changes dictionaries in place, so any existing keys in the data passed to `update` will have their old values discarded.



# Creating dictionaries from sequences

It's common to occasionally end up with two sequences that you want to pair up element-wise in a dictionary. As a first cut, you might write code like this:

```
mapping = {}  
for key, value in zip(key_list, value_list):  
    mapping[key] = value
```



Since a dictionary is essentially a collection of 2-tuples, the `dict` function accepts a list of 2-tuples:

```
In [104]: tuples = zip(range(5), reversed(range(5)))
```



```
In [105]: tuples
```

```
Out[105]: <zip at 0x17d604d00>
```

```
In [106]: mapping = dict(tuples)
```

```
In [107]: mapping
```


```
Out[107]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Later we'll talk about *dictionary comprehensions*, which are another way to construct dictionaries.

# Default Values

It's common to have logic like:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```



Thus, the dictionary methods `get` and `pop` can take a default value to be returned, so that the above `if-else` block can be written simply as:

```
value = some_dict.get(key, default_value)
```



# Default Values

`get` by default will return `None` if the key is not present, while `pop` will raise an exception. With setting values, it may be that the values in a dictionary are another kind of collection, like a list. For example, you could imagine categorizing a list of words by their first letters as a dictionary of lists:

```
In [108]: words = ["apple", "bat", "bar", "atom", "book"]
```

```
In [109]: by_letter = {}
```

```
In [110]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
.....:
```

```
In [111]: by_letter
```

```
Out[111]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

# Default Values

The `setdefault` dictionary method can be used to simplify this workflow. The preceding `for` loop can be rewritten as:

```
In [112]: by_letter = {}
```

```
In [113]: for word in words:
.....:     letter = word[0]
.....:     by_letter.setdefault(letter, []).append(word)
.....:
```

```
In [114]: by_letter
```

```
Out[114]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

# Default Values

The built-in `collections` module has a useful class, `defaultdict`, which makes this even easier. To create one, you pass a type or function for generating the default value for each slot in the dictionary:

```
In [115]: from collections import defaultdict
```

```
In [116]: by_letter = defaultdict(list)
```

```
In [117]: for word in words:
.....:     by_letter[word[0]].append(word)
```



# Valid Dictionary Key Types

While the values of a dictionary can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is *hashability*. You can check whether an object is hashable (can be used as a key in a dictionary) with the `hash` function:

```
In [118]: hash("string")
Out[118]: 4022908869268713487
```

```
In [119]: hash((1, 2, (2, 3)))
Out[119]: -9209053662355515447
```

```
In [120]: hash((1, 2, [2, 3])) # fails because lists are mutable
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-120-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'
```

# Valid Dictionary Key Types

The hash values you see when using the `hash` function in general will depend on the Python version you are using.

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can be:

```
In [121]: d = {}  
  
In [122]: d[tuple([1, 2, 3])] = 5  
  
In [123]: d  
Out[123]: {(1, 2, 3): 5}
```



# Sets

A *set* is an unordered collection of unique elements. A set can be created in two ways: via the `set` function or via a *set literal* with curly braces:

```
In [124]: set([2, 2, 2, 1, 3, 3])
```

```
Out[124]: {1, 2, 3}
```

```
In [125]: {2, 2, 2, 1, 3, 3}
```

```
Out[125]: {1, 2, 3}
```

Sets support mathematical *set operations* like union, intersection, difference, and symmetric difference. Consider these two example sets:

```
In [126]: a = {1, 2, 3, 4, 5}
```

```
In [127]: b = {3, 4, 5, 6, 7, 8}
```



# Sets

The union of these two sets is the set of distinct elements occurring in either set. This can be computed with either the `union` method or the `|` binary operator:

```
In [128]: a.union(b)
Out[128]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [129]: a | b
Out[129]: {1, 2, 3, 4, 5, 6, 7, 8}
```



The intersection contains the elements occurring in both sets. The `&` operator or the `intersection` method can be used:

```
In [130]: a.intersection(b)
Out[130]: {3, 4, 5}
```

```
In [131]: a & b
Out[131]: {3, 4, 5}
```



# Python Set Operations

Function	Alternative syntax	Description
<code>a.add(x)</code>	N/A	Add element <code>x</code> to set <code>a</code>
<code>a.clear()</code>	N/A	Reset set <code>a</code> to an empty state, discarding all of its elements
<code>a.remove(x)</code>	N/A	Remove element <code>x</code> from set <code>a</code>
<code>a.pop()</code>	N/A	Remove an arbitrary element from set <code>a</code> , raising <code>KeyError</code> if the set is empty
<code>a.union(b)</code>	<code>a   b</code>	All of the unique elements in <code>a</code> and <code>b</code>
<code>a.update(b)</code>	<code>a  = b</code>	Set the contents of <code>a</code> to be the union of the elements in <code>a</code> and <code>b</code>
<code>a.intersection(b)</code>	<code>a &amp; b</code>	All of the elements in <i>both</i> <code>a</code> and <code>b</code>

# Python Set Operations

<code>a.intersection_update(b)</code>	<code>a &amp;= b</code>	Set the contents of <code>a</code> to be the intersection of the elements in <code>a</code> and <code>b</code>
<code>a.difference(b)</code>	<code>a - b</code>	The elements in <code>a</code> that are not in <code>b</code>
<code>a.difference_update(b)</code>	<code>a -= b</code>	Set <code>a</code> to the elements in <code>a</code> that are not in <code>b</code>
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>	All of the elements in either <code>a</code> or <code>b</code> but <i>not both</i>
<code>a.symmetric_difference_update(b)</code>	<code>a ^= b</code>	Set <code>a</code> to contain the elements in either <code>a</code> or <code>b</code> but <i>not both</i>
<code>a.issubset(b)</code>	<code>&lt;=</code>	<code>True</code> if the elements of <code>a</code> are all contained in <code>b</code>
<code>a.issuperset(b)</code>	<code>&gt;=</code>	<code>True</code> if the elements of <code>b</code> are all contained in <code>a</code>
<code>a.isdisjoint(b)</code>	N/A	<code>True</code> if <code>a</code> and <code>b</code> have no elements in common

# Python Set Operations

All of the logical set operations have in-place counterparts, which enable you to replace the contents of the set on the left side of the operation with the result. For very large sets, this may be more efficient:

```
In [132]: c = a.copy()
```

```
In [133]: c |= b
```

```
In [134]: c
```

```
Out[134]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [135]: d = a.copy()
```

```
In [136]: d &= b
```

```
In [137]: d
```

```
Out[137]: {3, 4, 5}
```

# Python Set Operations

Like dictionary keys, set elements generally must be immutable, and they must be *hashable* (which means that calling `hash` on a value does not raise an exception). In order to store list-like elements (or other mutable sequences) in a set, you can convert them to tuples:

```
In [138]: my_data = [1, 2, 3, 4]
```

```
In [139]: my_set = {tuple(my_data)}
```

```
In [140]: my_set
```

```
Out[140]: {(1, 2, 3, 4)}
```

# Python Set Operations

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [141]: a_set = {1, 2, 3, 4, 5}
```

```
In [142]: {1, 2, 3}.issubset(a_set)
```

```
Out[142]: True
```

```
In [143]: a_set.issuperset({1, 2, 3})
```

```
Out[143]: True
```

Sets are equal if and only if their contents are equal:

```
In [144]: {1, 2, 3} == {3, 2, 1}
```

```
Out[144]: True
```

# Built-In Sequence Functions

## enumerate

It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
index = 0
for value in collection:
    # do something with value
    index += 1
```



Since this is so common, Python has a built-in function, `enumerate`, which returns a sequence of `(i, value)` tuples:

```
for index, value in enumerate(collection):
    # do something with value
```



# Built-In Sequence Functions

## sorted

The `sorted` function returns a new sorted list from the elements of any sequence:

```
In [145]: sorted([7, 1, 2, 6, 0, 3, 2])
```

```
Out[145]: [0, 1, 2, 2, 3, 6, 7]
```

```
In [146]: sorted("horse race")
```

```
Out[146]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

The `sorted` function accepts the same arguments as the `sort` method on lists.



# Built-In Sequence Functions

## zip

`zip` “pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples:

```
In [147]: seq1 = ["foo", "bar", "baz"]
```

```
In [148]: seq2 = ["one", "two", "three"]
```

```
In [149]: zipped = zip(seq1, seq2)
```

```
In [150]: list(zipped)
```

```
Out[150]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

# Built-In Sequence Functions

`zip` can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [151]: seq3 = [False, True]

In [152]: list(zip(seq1, seq2, seq3))
Out[152]: [('foo', 'one', False), ('bar', 'two', True)]
```

A common use of `zip` is simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:

```
In [153]: for index, (a, b) in enumerate(zip(seq1, seq2)):
.....:     print(f"{index}: {a}, {b}")
.....:
0: foo, one
1: bar, two
2: baz, three
```

# Built-In Sequence Functions

## reversed

`reversed` iterates over the elements of a sequence in reverse order:

```
In [154]: list(reversed(range(10)))  
Out[154]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```



Keep in mind that `reversed` is a generator (to be discussed in some more detail later), so it does not create the reversed sequence until materialized (e.g., with `list` or a `for` loop).

# List, Set, and Dictionary Comprehensions

*List comprehensions* are a convenient and widely used Python language feature. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter into one concise expression. They take the basic form:

```
[expr for value in collection if condition]
```



This is equivalent to the following `for` loop:

```
result = []  
for value in collection:  
    if condition:  
        result.append(expr)
```



# List, Set, and Dictionary Comprehensions

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and convert them to uppercase like this:

```
In [155]: strings = ["a", "as", "bat", "car", "dove", "python"]
```

```
In [156]: [x.upper() for x in strings if len(x) > 2]
```

```
Out[156]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dictionary comprehensions are a natural extension, producing sets and dictionaries in an idiomatically similar way instead of lists.

A dictionary comprehension looks like this:

```
dict_comp = {key-expr: value-expr for value in collection  
              if condition}
```

# List, Set, and Dictionary Comprehensions

A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets:

```
set_comp = {expr for value in collection if condition}
```



Like list comprehensions, set and dictionary comprehensions are mostly conveniences, but they similarly can make code both easier to write and read. Consider the list of strings from before. Suppose we wanted a set containing just the lengths of the strings contained in the collection; we could easily compute this using a set comprehension:

```
In [157]: unique_lengths = {len(x) for x in strings}
```



```
In [158]: unique_lengths
```

```
Out[158]: {1, 2, 3, 4, 6}
```

# List, Set, and Dictionary Comprehensions

We could also express this more functionally using the `map` function, introduced shortly:

```
In [159]: set(map(len, strings))  
Out[159]: {1, 2, 3, 4, 6}
```



As a simple dictionary comprehension example, we could create a lookup map of these strings for their locations in the list:

```
In [160]: loc_mapping = {value: index for index, value in enumerate(strings)}  
  
In [161]: loc_mapping  
Out[161]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```



# List, Set, and Dictionary Comprehensions

## Nested list comprehensions

Suppose we have a list of lists containing some English and Spanish names:

```
In [162]: all_data = [ ["John", "Emily", "Michael", "Mary", "Steven"],  
.....:                ["Maria", "Juan", "Javier", "Natalia", "Pilar"] ]
```

Suppose we wanted to get a single list containing all names with two or more **a**'s in them. We could certainly do this with a simple **for** loop:

```
In [163]: names_of_interest = []  
  
In [164]: for names in all_data:  
.....:     enough_as = [name for name in names if name.count("a") >= 2]  
.....:     names_of_interest.extend(enough_as)  
.....:  
  
In [165]: names_of_interest  
Out[165]: ['Maria', 'Natalia']
```



# List, Set, and Dictionary Comprehensions

You can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [166]: result = [name for names in all_data for name in names
.....:               if name.count("a") >= 2]

In [167]: result
Out[167]: ['Maria', 'Natalia']
```

At first, nested list comprehensions are a bit hard to wrap your head around. The `for` parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before. Here is another example where we “flatten” a list of tuples of integers into a simple list of integers:

```
In [168]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [169]: flattened = [x for tup in some_tuples for x in tup]

In [170]: flattened
Out[170]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# List, Set, and Dictionary Comprehensions

Keep in mind that the order of the `for` expressions would be the same if you wrote a nested `for` loop instead of a list comprehension:

```
flattened = []  
  
for tup in some_tuples:  
    for x in tup:  
        flattened.append(x)
```

You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting, you should probably start to question whether this makes sense from a code readability standpoint. It's important to distinguish the syntax just shown from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [172]: [[x for x in tup] for tup in some_tuples]  
Out[172]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

This produces a list of lists, rather than a flattened list of all of the inner elements.

# Functions

- 1 **Primary Role in Code Organization:** Functions represent the primary method for organizing and structuring code in Python, playing a crucial role in maintaining clean and manageable codebases.
- 2 **Code Reusability:** Functions are pivotal for code reusability. They are particularly useful when you need to repeat similar code multiple times, making them a key tool for avoiding redundancy.
- 3 **Criterion for Function Creation:** A good rule of thumb is to create a function when you anticipate the need to repeat the same or similar code more than once.
- 4 **Enhancing Readability:** Functions enhance code readability by encapsulating a group of Python statements under a descriptive name, thereby making the code more understandable.
- 5 **Naming Conventions:** The naming of functions is an important aspect, as a well-chosen name can convey the purpose and functionality of the code block it encapsulates.

# Functions

Functions are declared with the `def` keyword. A function contains a block of code with an optional use of the `return` keyword:

```
In [173]: def my_function(x, y):  
.....:     return x + y
```

When a line with `return` is reached, the value or expression after `return` is sent to the context where the function was called, for example:

```
In [174]: my_function(1, 2)  
Out[174]: 3
```

```
In [175]: result = my_function(1, 2)
```

```
In [176]: result  
Out[176]: 3
```

# Functions

There is no issue with having multiple `return` statements. If Python reaches the end of a function without encountering a `return` statement, `None` is returned automatically. For example:

```
In [177]: def function_without_return(x):
.....:     print(x)

In [178]: result = function_without_return("hello!")
hello!

In [179]: print(result)
None
```

Each function can have *positional* arguments and *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. Here we will define a function with an optional `z` argument with the default value `1.5`:

```
def my_function2(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

# Functions

While keyword arguments are optional, all positional arguments must be specified when calling a function.

You can pass values to the `z` argument with or without the keyword provided, though using the keyword is encouraged:

```
In [181]: my_function2(5, 6, z=0.7)
```

```
Out[181]: 0.06363636363636363
```

```
In [182]: my_function2(3.14, 7, 3.5)
```

```
Out[182]: 35.49
```

```
In [183]: my_function2(10, 20)
```

```
Out[183]: 45.0
```

The main restriction on function arguments is that the keyword arguments *must* follow the positional arguments (if any). You can specify keyword arguments in any order. This frees you from having to remember the order in which the function arguments were specified. You need to remember only what their names are.

# Namespaces, Scope, and Local Functions

Functions can access variables created inside the function as well as those outside the function in higher (or even *global*) scopes. An alternative and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and is immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed (with some exceptions that are outside the purview of this chapter). Consider the following function:

```
def func():  
    a = []  
    for i in range(5):  
        a.append(i)
```



# Namespaces, Scope, and Local Functions

When `func()` is called, the empty list `a` is created, five elements are appended, and then `a` is destroyed when the function exits. Suppose instead we had declared `a` as follows:

```
In [184]: a = []  
  
In [185]: def func():  
.....:     for i in range(5):  
.....:         a.append(i)
```

Each call to `func` will modify list `a`:

```
In [186]: func()  
  
In [187]: a  
Out[187]: [0, 1, 2, 3, 4]  
  
In [188]: func()  
  
In [189]: a  
Out[189]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```



# Namespaces, Scope, and Local Functions

Assigning variables outside of the function's scope is possible, but those variables must be declared explicitly using either the `global` or `nonlocal` keywords:

```
In [190]: a = None

In [191]: def bind_a_variable():
.....:     global a
.....:     a = []
.....:     bind_a_variable()
.....:

In [192]: print(a)
[]
```

`nonlocal` allows a function to modify variables defined in a higher-level scope that is not global. Since its use is somewhat esoteric (I never use it in this book), I refer you to the Python documentation to learn more about it.

# Functions Are Objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

```
In [193]: states = ["  Alabama ", "Georgia!", "Georgia", "georgia", "FlOrIda",  
.....:           "south  carolina##", "West virginia?"]
```

Anyone who has ever worked with user-submitted survey data has seen messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: stripping whitespace, removing punctuation symbols, and standardizing proper capitalization. One way to do this is to use built-in string methods along with the `re` standard library module for regular expressions:

# Functions Are Objects

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub("[!#?]", "", value)
        value = value.title()
        result.append(value)
    return result
```

The result looks like this:

```
In [195]: clean_strings(states)
Out[195]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

# Functions Are Objects

An alternative approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings:

```
def remove_punctuation(value):  
    return re.sub("[!#?]", "", value)  
  
clean_ops = [str.strip, remove_punctuation, str.title]  
  
def clean_strings(strings, ops):  
    result = []  
    for value in strings:  
        for func in ops:  
            value = func(value)  
        result.append(value)  
    return result
```



# Functions Are Objects

Then we have the following:

```
In [197]: clean_strings(states, clean_ops)
Out[197]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```



# Functions Are Objects

A more *functional* pattern like this enables you to easily modify how the strings are transformed at a very high level. The `clean_strings` function is also now more reusable and generic.

You can use functions as arguments to other functions like the built-in `map` function, which applies a function to a sequence of some kind:

```
In [198]: for x in map(remove_punctuation, states):  
.....:     print(x)  
Alabama  
Georgia  
Georgia  
georgia  
Fl0rIda  
south carolina  
West virginia
```

`map` can be used as an alternative to list comprehensions without any filter.

# Anonymous (Lambda) Functions

Python has support for so-called *anonymous* or *lambda* functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the `lambda` keyword, which has no meaning other than “we are declaring an anonymous function”:

```
In [199]: def short_function(x):  
.....:     return x * 2  
  
In [200]: equiv_anon = lambda x: x * 2
```



# Anonymous (Lambda) Functions

I usually refer to these as lambda functions in the rest of the book. They are especially convenient in data analysis because, as you'll see, there are many cases where data transformation functions will take functions as arguments. It's often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable. Consider this example:

```
In [201]: def apply_to_list(some_list, f):  
.....:     return [f(x) for x in some_list]
```

```
In [202]: ints = [4, 0, 1, 5, 6]
```

```
In [203]: apply_to_list(ints, lambda x: x * 2)  
Out[203]: [8, 0, 2, 10, 12]
```



# Anonymous (Lambda) Functions

You could also have written `[x * 2 for x in ints]`, but here we were able to succinctly pass a custom operator to the `apply_to_list` function.

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

```
In [204]: strings = ["foo", "card", "bar", "aaaa", "abab"]
```

Here we could pass a lambda function to the list's `sort` method:

```
In [205]: strings.sort(key=lambda x: len(set(x)))
```

```
In [206]: strings
```

```
Out[206]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

# Generators

Many objects in Python support iteration, such as over objects in a list or lines in a file. This is accomplished by means of the *iterator protocol*, a generic way to make objects iterable. For example, iterating over a dictionary yields the dictionary keys:

```
In [207]: some_dict = {"a": 1, "b": 2, "c": 3}
```

```
In [208]: for key in some_dict:
```

```
.....:     print(key)
```

```
a
```

```
b
```

```
c
```

When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of `some_dict`:

```
In [209]: dict_iterator = iter(some_dict)
```

```
In [210]: dict_iterator
```

```
Out[210]: <dict_keyiterator at 0x17d60e020>
```

# Generators

An iterator is any object that will yield objects to the Python interpreter when used in a context like a `for` loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [211]: list(dict_iterator)
Out[211]: ['a', 'b', 'c']
```

A *generator* is a convenient way, similar to writing a normal function, to construct a new iterable object. Whereas normal functions execute and return a single result at a time, generators can return a sequence of multiple values by pausing and resuming execution each time the generator is used. To create a generator, use the `yield` keyword instead of `return` in a function:

```
def squares(n=10):
    print(f"Generating squares from 1 to {n ** 2}")
    for i in range(1, n + 1):
        yield i ** 2
```

# Generators

When you actually call the generator, no code is immediately executed:

```
In [213]: gen = squares()
```

```
In [214]: gen
```

```
Out[214]: <generator object squares at 0x17d5fea40>
```

It is not until you request elements from the generator that it begins executing its code:

```
In [215]: for x in gen:
```

```
.....:     print(x, end=" ")
```

```
Generating squares from 1 to 100
```

```
1 4 9 16 25 36 49 64 81 100
```

# itertools module

The standard library `itertools` module has a collection of generators for many common data algorithms. For example, `groupby` takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function. Here's an example:

```
In [220]: import itertools

In [221]: def first_letter(x):
.....:     return x[0]

In [222]: names = ["Alan", "Adam", "Wes", "Will", "Albert", "Steven"]

In [223]: for letter, names in itertools.groupby(names, first_letter):
.....:     print(letter, list(names)) # names is a generator
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

# SEAS 6414 – Python Applications in Data Analytics

---

**THE GEORGE  
WASHINGTON  
UNIVERSITY**

---

WASHINGTON, DC

Dr. Adewale Akinfaderin

Lecture 2 - Spring 2024