PYTHON BASICS

There are a few key aspects of python that are incredibly useful for working with big data and big data applications. In this lecture, we will cover the basics to ensure everyone has a good understanding of python development going forward.

[**Python Basics**](#_jkz21jr4sfqx) **3**

[Lists](#_mgttmkbj4t60) 3

[Creating lists](#_7x0pg6eazeyb) 4

[Accessing Elements in the List](#_mzp3g3wr5oim) 5

[Selecting Ranges](#_kjloyd5o1dq) 7

[Editing List Content](#_ifabi9za8qpk) 7

[Add Items to a List](#_2r9ixtepz4st) 7

[.append() vs. .extend()](#_oz0wc33y9mg3) 8

[.insert(index, value)](#_bt6ietjhi5aj) 8

[Remove Items from a List](#_wnlsaid4vd12) 9

[.pop()](#_7dsv60dszk48) 9

[.pop(index)](#_1c9vxemcjy6z) 9

[Copying Lists](#_tgq7y1aj4kd7) 10

[Update/Replace Items in a List](#_i388jabxg5cv) 11

[Join Items](#_h30gnn42v9mw) 12

[Split Items](#_onzan334phcy) 13

[Built-in Operators for Analyzing Lists](#_vjsu8cj0os11) 13

[.index()](#_kkawfqyppke4) 14

[.count()](#_mf0n7r3qfmqn) 14

[Sorting Lists](#_mhy2elvr6xx) 15

[.sort()](#_5nz56rvani8d) 15

[sorted()](#_c5qqc6v1klak) 16

[Key Takeaways](#_z4hfr4moace9) 17

[Loops](#_e9dteey6i08w) 18

[Iterating with Loops](#_p1zstzbgqik7) 18

[the while loop](#_7hylco9h4fjk) 20

[Counting](#_z7as3r2k01fe) 20

[Booleans](#_sjtn099olbui) 21

[the for loop](#_p7udthqm6db0) 22

[Loops with Ranges](#_6yxo4ws0bg2z) 22

[range()](#_j4r1rk21c257) 22

[range() with a while loop](#_asolmqmfspys) 22

[range() with a for loop](#_2buoxg4i5a86) 23

[enumerate()](#_h1df5unufi3t) 23

[Control Flow with break, continue](#_9bmpy2l9xl9k) 24

[The break Keyword](#_zcngq412idq2) 24

[The continue Keyword](#_dveeznkaiexy) 26

[Infinite Loop Example](#_nelw7vxbggzp) 27

[Dicts](#_uaqzdl72b3se) 27

[Intro](#_bgdipctspb4j) 27

[Dict Structure](#_gto4626lm78d) 28

[Creating Dicts](#_3cobuubqb57f) 29

[Method 1: Pass in key value pairs directly using {}:](#_u340rg9knr8x) 29

[Method 2: Create an empty dict with the dict() method.](#_8zp2txn3uqn7) 29

[Method 3: Convert a list of tuples into a dict using dict()](#_2h6wtg1ik1xg) 30

[Method 4: Use zip() to convert two lists into dict keys and values](#_74ir1cz4g24e) 30

[Accessing Dict Data](#_u0n87yc8241s) 30

[Referencing Values by Keys](#_l4mrtcnrodg6) 31

[Retrieving All Keys, Values, & Key/Value Pairs](#_8dhkcnw9zoxs) 32

[Isolating Keys & Values](#_a268c0lgf9uc) 32

[Isolating Key/Value Pairs](#_1dplebo3mi0t) 34

[Built-in Operators for Manipulating Dicts](#_89015jip4ec5) 35

[Add or Edit Dict Items](#_dp3t7bpjsdsp) 35

[Remove Items from a Dict](#_uqqp1r9mq4rt) 37

[.clear() simply empties the dict of all items.](#_9pzjx2vn0s4u) 37

[.pop(key, value):](#_9smegckxtm7y) 37

[popitem():](#_syafzpamdq6a) 38

[Iterating Through Dicts](#_nxp660w8f2x0) 38

[Iterate Through Dict Items](#_fp9vgddmm4ja) 39

[Iterate Through Dict Keys](#_wsp77bfxpto2) 39

[Sorting Dicts with Loops](#_o4zekbfqf9cx) 40

[By Key](#_gwhbe5wvgv3f) 40

[By the Values of Each Key](#_nuzq7q57nbot) 40

[Key Takeaways](#_n7axr5pcyso4) 41

[Functions](#_375jsiuoe0ap) 42

[Anatomy of a Function](#_lq9vn2ck4tap) 43

[Naming & Input](#_qldb8v9ybs69) 43

[Function Body](#_fee1k5i03yb7) 43

[Output](#_qymivjzefg08) 43

[A Simple First Function](#_d4ac2bn5q1a3) 43

[Input the number we want to square](#_fpqxy3gf9z61) 44

[Calculate the square of that number](#_g0uuxm1smza9) 44

[Output the square of that number](#_gsep0la6pqij) 44

[Types of Function Input](#_udqlvpau8l68) 44

[Required vs. Optional Parameters](#_ds7sizv5yw9b) 44

[Default Input Values](#_nc7c6ku4g6ar) 46

[Variable number of Arguments](#_22bn07e7t0vi) 48

[Variable Scope Recap](#_en8taegeq03n) 49

[Classes & Inheritance](#_chiry6h5o81n) 50

[High-Level Overview](#_s107x5fv6hqc) 51

[Creating & Structuring Classes](#_kkigkkt20c9u) 51

[Attributes & Methods](#_96g6dr1fzsen) 51

[Inheritance Basics](#_i4dyf6ofvtwv) 52

[The init() Method & the self Keyword](#_xpt5s8c3ufsw) 53

[Class vs. Instance Variables](#_szc7csi9pwym) 54

[The init() Method & the self Keyword](#_7w0ahvi4v76r) 54

[Class Definition Example 1 - Basic Elements in Context](#_isqkxhw8aal) 55

[Class Definition Example 2 - Child Classes & Inheritance](#_975i8bibbxsv) 56

[Class Instantiation & Modification](#_sin20qxvz3wv) 57

[Checking Class Values](#_44gf3grtxukk) 58

[Key Takeaways](#_6axqr4q7ee0v) 59

# 

# 

# 

# 

# 

# 

# 

# 

# Python Basics

This section is intended for review. Feel free to skip or skim.

## Lists

In order to begin to truly write dynamic programs, we need to be able to work with data even when we do not know how much we have. Since *variables hold only one item*, Python has some objects that hold collections of data. A **list** object is one of those. **Lists** hold multiple items, called **elements**. List elements can represent any data type, and most, importantly, *a single list can hold any mix of different data types,* ***including other lists****.*

### **Creating lists**

To declare a new list varaible, you have two options -- [] or list(). The [] syntax is a bit more straightforward. If you choose to use the list() method, you still have to pass the items to it within [] because it will only accept **one** parameter...

|  |
| --- |
| # Empty empty1 = [] empty2 = list()  # Mixed Data Types mix1 = [**True**, ['seal', 'spider monkey'], 22, 'lion', [**False**, 13]] mix2 = list([**True**, ['seal', 'spider monkey'], 22, 'lion', [**False**, 13]])  # Nested List nested1 = [['circus', 'clown'], ['trapeze', 'artist']] nested2 = list([['circus', 'clown'], ['trapeze', 'artist']]) |

The list() method can be tricky. Here's what happens if you pass more than one item to it:

|  |
| --- |
| colors = list('red', 'yellow', 'green') ### TypeError: list() takes at most 1 argument (3 given) |

Even if you do pass only *one* item, you still have to enclose it with []. If you pass a single number or a single boolean without using brackets, you'll get a TypeError.

|  |
| --- |
| x = list(100) # or any float e.g. 100.0 ### TypeError: 'int' object is not iterable  y = list(**True**) # booleans ### TypeError: 'bool' object is not iterable |

And if you pass a string without using brackets, you won't get quite what you want.

|  |
| --- |
| z = list('Layla') print(z) # ['L', 'a', 'y', 'l', 'a'] |

And if you think it works for sentences by parsing a new list element at each space character...*nope!*

|  |
| --- |
| sentence = list('I like to sing!') print(sentence) # ['I', ' ', 'l', 'i', 'k', 'e', ' ', 't', 'o', ' ', 's', 'i', 'n', 'g', '!'] |

### **Accessing Elements in the List**

The **list index** means the location of each element in the list. List indexes start counting at 0!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **List** | **'snickerdoodles'** | **'shortbread'** | **'oatmeal raisin'** | **'gingersnaps'** | **'macarons'** |
| Index | 0 | 1 | 2 | 3 | 4 |

To access one of them use the syntax list\_name[index\_position].

|  |
| --- |
| cookies = ['snickerdoodles', 'shortbread', 'oatmeal raisin', 'gingersnaps', 'macarons'] print(cookies[0]) # snickerdoodles print(cookies[1]) # shortbread print(cookies[4]) # macarons |

What if you want to grab a single element from a list nested within a list though? Simply add another level of index selection.

|  |
| --- |
| nested\_lists = list([['circus', 'clown'], ['trapeze', 'artist']]) print(nested\_lists[0][0]) # circus print(nested\_lists[1][0]) # trapeze |

The len() function will give you the total number of list elements (regardless of data type).

|  |
| --- |
| cookies = ['snickerdoodles', 'shortbread', 'oatmeal raisin', 'gingersnaps', 'macarons'] types\_of\_cookies = len(cookies) print(types\_of\_cookies) # 5 |

**REMEMBER!** Because the index starts at 0, the index position of the last item in the list will NOT be equal to the list's length.

If you pass in any list index greater than or equal to the length of the list, you will get an IndexError:

|  |
| --- |
| print(cookies[types\_of\_cookies]) ### IndexError: list index out of range |

Aside from merely finding the length, len() comes in handy if you need to dynamically select the last element of a list. Logically, the length of the list minus 1 will give you the index of the last list element, i.e. **index\_of\_last\_element = len(list\_name) - 1**.

|  |
| --- |
| cookies = ['snickerdoodles', 'shortbread', 'oatmeal raisin', 'gingersnaps', 'macarons'] types\_of\_cookies = len(cookies)  # (types\_of\_cookies - 1) is the index of the last element print(cookies[types\_of\_cookies - 1]) # Dasha |

That brings us to something called **negative indexing**. Because of the above rule, Python allows you to take this shortcut to access the last list element:

|  |
| --- |
| print(cookies[-1]) |

Likewise, as you count backwards from the last list element, the negative index extends...

|  |
| --- |
| print(f'2nd to last item is {cookies[-2]}') print(f'3rd to last item is {cookies[-3]}') |

#### **Selecting Ranges**

To select multiple items from a list, simply pass the range of indeces which hold the desired elements, e.g. my\_list[2:7]. It's important to remember that **the upper bound is NOT inclusive**. In other words, if you want the elements at index 3 and index 4, you have to write [3:5]. (Likewise, my\_list[1:1] would print nothing.)

|  |
| --- |
| cookies = ['snickerdoodle', 'chocolate chip', 'shortbread', 'oatmeal raisin', 'gingerbread', 'pizelle', 'macarons', 'gingersnaps'] print(cookies[3:5]) |

Omitting one or both bounds acts as a shortcut when the length and order of your list might vary. It allows you to say, "Give me everything from the beginning of the list to index x" or "Give me everything from index x and onward."

|  |
| --- |
| cookies = ['snickerdoodle', 'chocolate chip', 'shortbread', 'oatmeal raisin', 'gingerbread', 'pizelle', 'macarons', 'gingersnaps']  print('[:2] -- ', my\_class[:2]) # All indeces up to, but NOT including index 2  print('[2:] -- ', my\_class[2:]) # Index 2 through end of list |

### **Editing List Content**

#### **Add Items to a List**

If you want to add elements to a list, you can use any of the below methods:

* .append(): adds items to the end of a list in one chunk
* .extend(): adds items to the end of a list individually
* .insert(index, value): adds an item to a specific position in the list

#### **.append() vs. .extend()**

Let's go through some examples. First, we'll look at the difference between .append() and .extend(). As mentioned above, .append() will add whatever value or group of values you pass it *in one chunk*. In contrast, if you pass a group of values into .extend(), it will add *each* element of the group *individually*.

**APPEND**

|  |
| --- |
| pies = ['apple', 'pumpkin', 'pecan', 'blueberry'] more\_pies = ['lemon meringue', 'strawberry peach', 'banana cream']  pies.append(more\_pies) print(f'.append(): {pies}\n') |

**EXTEND**

|  |
| --- |
| pies = ['apple', 'pumpkin', 'pecan', 'blueberry'] more\_pies = ['lemon meringue', 'strawberry peach', 'banana cream']  pies.extend(more\_pies) print(f'.extend(): {pies}') |

In the next example, take a look at how .extend() only considers individual values of the *parent list* passed in. It still adds the nested lists - ['f', 'g'] and ['h', 'i'] - to our list x as their own items.

|  |
| --- |
| x = ['a', 'b', 'c', 'd'] y = ['e', ['f', 'g'], ['h', 'i'], 'j'] x.extend(y) print(x) # ['a', 'b', 'c', 'd', 'e', ['f', 'g'], ['h', 'i'], 'j'] |

#### **.insert(index, value)**

If you want to add an item to a specific point in your list, you can pass the desired index and value into .insert() as follows.

|  |
| --- |
| pies = ['apple', 'pumpkin', 'pecan', 'blueberry'] pies.insert(1, 'key lime') print(pies) |

Whatever you pass into the value parameter will be added as a single element though!

|  |
| --- |
| pies = ['apple', 'pumpkin', 'pecan', 'blueberry'] pies.insert(1, ['lemon meringue', 'strawberry peach']) print(pies) |

#### **Remove Items from a List**

Likewise, you can use .pop() or .pop(index) to remove any type of element from a list.

#### **.pop()**

* Removes an item from the end of the list.

|  |
| --- |
| pies = ['cherry', 'lemon meringue', 'pecan', 'key lime', 'blackberry', 'strawberry peach', 'blueberry'] eaten = pies.pop() print(f'The {eaten} pie got eaten.') print(pies) |

#### **.pop(index)**

* Removes an item from the list.
* Can take an index.

|  |
| --- |
| pies = ['cherry', 'lemon meringue', 'pecan', 'key lime', 'blackberry', 'strawberry peach'] eaten = pies.pop(2) # Remember to count from 0! print(f'The {eaten} pie got eaten.') |

#### **Copying Lists**

You can create a copy of a list simply by declaring a new variable and setting it equal to the existing list, as seen with pies2 below. In contrast, you can create a *deep copy*, one that doesn't point back to the original, using [:] as seen with pies3 below.

|  |
| --- |
| pies1 = ['strawberry', 'cherry', 'blackberry', 'blueberry'] pies2 = pies1 pies3 = pies1[:]  print(f''' pies1: {pies1} pies2: {pies2} pies3: {pies3} ''') |

Let's look at the difference in action. With the first method, you're saying that pies2 points to the place in memory where pies1 is stored. That means if you edit pies2, you're effectively editing pies1 along with it.

|  |
| --- |
| pies1 = ['strawberry', 'cherry', 'blackberry', 'blueberry'] pies2 = pies1  print(f''' BEFORE: pies1: {pies1} pies2: {pies2} ''')  pies2.pop()  print(f''' AFTER pies2.pop(): pies1: {pies1} pies2: {pies2} ''') |

A deep copy creates a *separate instance* of the list in the program's memory so that the two are not tied together at all. Editing a deep copy like pies3 will NOT alter pies1.

|  |
| --- |
| pies1 = ['strawberry', 'cherry', 'blackberry', 'blueberry'] pies3 = pies1[:]  print(f''' BEFORE: pies1: {pies1} pies2: {pies3} ''')  pies3.pop()  print(f''' AFTER pies3.pop(): pies1: {pies1} pies3: {pies3} ''') |

#### **Update/Replace Items in a List**

To replace items in a list, you reference them by their index position and simply declare a new value. The general syntax is my\_list[<index>] = <value>.

|  |
| --- |
| pies = ['apple', 'pumpkin', 'pecan', 'blueberry'] pies[0] = 'banana cream' print(pies) |

Let's say you specify one index position and pass a group of elements like the below. That group *as a whole* replaces the element at the specified index position.

|  |
| --- |
| pies = ['banana cream', 'pumpkin', 'pecan', 'blueberry'] pies[1] = ['key lime', 'lemon meringue'] print(pies) |

If you specify a range of index positions, each item in the group gets added individually (sort of like .extend()). Here are a few examples...

* This replaces everything from index position 1 onward with the three new elements passed.

|  |
| --- |
| pies = ['strawberry', 'banana cream', 'key lime'] pies[1:] = ['blueberry', 'cherry', 'blackberry'] print(pies) # ['strawberry', 'blueberry', 'cherry', 'blackberry'] |

* This replaces every element up to, but NOT including index position 2 with the three new elements passed. Notice how the *three* new elements replaced only *two* existing elements.

|  |
| --- |
| pies = ['strawberry', 'blueberry', 'cherry', 'blackberry'] pies[:2] = ['apple', 'pumpkin', 'pecan'] print(pies) # ['apple', 'pumpkin', 'pecan', 'cherry', 'blackberry'] |

* Even if you pass one new element, it must be in list format. Otherwise, Python will try to break it up into a list like the below...

|  |
| --- |
| pies1 = ['strawberry', 'banana cream', 'key lime', 'apple', 'pumpkin'] pies2 = pies1[:]  pies1[1:3] = ['blueberry'] pies2[1:3] = 'blueberry' print(pies1) # ['strawberry', 'blueberry', 'apple', 'pumpkin'] print(pies2) # ['strawberry', 'b', 'l', 'u', 'e', 'b', 'e', 'r', 'r', 'y','apple', 'pumpkin'] |

#### **Join Items**

If you need to, you can compile your list items into a single string using .join(). You CANNOT do this with lists of numbers, booleans, or a list containing a mix of strings and other data types though.

|  |
| --- |
| letters = ['j', 'u', 'l', 'i', 'a', 'n', 'n', 'a'] name = ''.join(letters) print(name) # 'julianna'  words = ['this', 'is', 'fun'] sentence = ' '.join(words) print(f'{sentence}.') # 'this is fun.' |

#### **Split Items**

Conversely, you can separate a string using .split(<by\_char>), which will parse values out of a string and turn each value into a list item. This one doesn't work for single words you might want to split into individual characters. That said, you *can* specify what character should convey to the method when to split out a new item. By default, .split(<by\_char>) will use a space character to split the string.

|  |
| --- |
| x = 'this is fun' sentence = x.split() # note - using default split char at space print(sentence) # ['this', 'is', 'fun']  y = 'Sandra,hi@email.com,646-212-1234,8 Cherry Lane,Splitsville,FL,58028' data = y.split(',') print(data) # ['Sandra', 'hi@email.com', '646-212-1234', '8 Cherry Lane', 'Splitsville', 'FL', '58028'] |

### **Built-in Operators for Analyzing Lists**

Python has some built-in operations that allow you to analyze the content of a list. Some basic ones include:

**sum()**: returns the sum of all items in *numerical lists*.

|  |
| --- |
| team\_batting\_avgs = [.328, .299, .208, .301, .275, .226, .253, .232, .287] sum\_avgs = sum(team\_batting\_avgs) print(f'The total of all the batting averages is {sum\_avgs}.') |

**min()** & **max()**: return the smallest or largest number *in a numerical list*.

|  |
| --- |
| team\_batting\_avgs = [.299, .208, .301, .275, .328, .226, .253, .232, .287]  print(f'Highest: {max(team\_batting\_avgs)}') # .328 print(f'Lowest: {min(team\_batting\_avgs)}') # .208 |

#### **.index()**

Given a list element's value, return its index.

|  |
| --- |
| sentence = ['a', 'purple', 'pig', 'and', 'a', 'green', 'donkey', 'flew', 'a', 'kite', 'in', 'the', 'middle','of', 'the', 'night', 'and', 'ended', 'up', 'sunburnt']  pig\_index = sentence.index('pig') print(pig\_index) |

If the element occurs multiple times in the list, .index() will only return the index of its *first* occurrence!

|  |
| --- |
| sentence = ['a', 'purple', 'pig', 'AND', 'a', 'green', 'donkey', 'flew', 'a', 'kite', 'in', 'the', 'middle','of', 'the', 'night', 'AND', 'ended', 'up', 'sunburnt']  and\_index = sentence.index('AND') print(and\_index) # 3 |

#### **.count()**

This returns the number of occurrences of a *single, distinct element* within a list. That means for example, if you're searching a list of words (i.e. strings) for the occurrences of a single letter, only instances where that single letter appears as its own list item will be counted.

|  |
| --- |
| nums = [84, 8, 18, 8, 28, 6, 10, 8, 78, 9] print(nums.count(8)) # 3   sentence = ['a', 'purple', 'pig', 'and', 'a', 'green', 'donkey', 'flew', 'a', 'kite', 'in', 'the', 'middle','of', 'the', 'night', 'and', 'ended', 'up', 'sunburnt'] print(sentence.count('a')) # 3 |

If you want to count the occurrences of a single character within a larger value, you can still use .count()...

|  |
| --- |
| word = 'sunburnt' print(word.count('u')) # 2 |

**GOTCHA!** .count() throws an error if you try to count the number of times the digit "2" appears in the number below. And remember, you CANNOT use the .split() method as a workaround because .split() only works on strings!

|  |
| --- |
| num = 22384232348 print(num.count(2)) # will throw error |

### **Sorting Lists**

If you want to organize your lists better, you can sort them with the .sort() or sorted() functions. You can sort:

* Numbers: ascending and descending order
* Strings: alphabetically and reverse alphabetically
* You **cannot** sort a list that includes different data types.

**GOTCHA!** It's important to remember that the .sort() method modifies the list *in place*, while the sorted() function requires you to assign its result back to the variable.

#### **.sort()**

The first two examples below illustrate how to sort lists in ascending order/alphabetically using .sort().

|  |
| --- |
| numbers = [1, 3, 7, 5, 2, 4, 6] numbers.sort() print(numbers) # [1, 2, 3, 4, 5, 6, 7]   letters = ['b', 'e', 'c', 'a', 'd'] letters.sort() print(letters) # ['a', 'b', 'c', 'd', 'e'] |

To do this in descending order, simply add reverse=True as an argument in .sort() like this:

|  |
| --- |
| numbers = [1, 3, 7, 5, 2, 4, 6] numbers.sort(reverse = **True**) print(numbers) # [7, 6, 5, 4, 3, 2, 1]  letters = ['b', 'e', 'c', 'a', 'd'] letters.sort(reverse = **True**) print(letters) # ['e', 'd', 'c', 'b', 'a'] |

#### **sorted()**

The first two examples below illustrate how to sort lists in **ascending order**/alphabetically using sorted().

|  |
| --- |
| numbers = [1, 3, 7, 5, 2, 4, 6] ascending = sorted(numbers) print(ascending) # [1, 2, 3, 4, 5, 6, 7]   letters = ['b', 'e', 'c', 'a', 'd'] ascending = sorted(letters) print(ascending) # ['a', 'b', 'c', 'd', 'e'] |

To do this in **descending order**, simply add reverse=True as an argument in sorted() like this:

|  |
| --- |
| numbers = [1, 3, 7, 5, 2, 4, 6] descending = sorted(numbers, reverse=**True**) print(descending) # [7, 6, 5, 4, 3, 2, 1]   letters = ['b', 'e', 'c', 'a', 'd'] descending = sorted(letters, reverse=**True**) print(descending) # ['e', 'd', 'c', 'b', 'a'] |

### **Key Takeaways**

* Pass comma-separated values within [] to create a list object
* An element's *index position* refers to its numerical place in the list. The index always starts at 0, i.e. the first element of the list is at index position 0.
* Access a list elements based on its index position using the syntax my\_list[<index>]
* len() will return the length of the list
* .append() adds an element to the end of a list
  + If you pass a group of elements to .append(), they get added as a *single* element.
* To elements from a group to a list individually, pass the group to .extend()
* .pop() removes the last item from the list
  + Pass an index position into .pop() to remove a specific element
* my\_list[idx] = new\_value updates an element at a specific index position
* To create a deep, or completely separate, copy, use my\_list[:]
* For numerical lists...
  + Use sum() to find the total of all the numbers in the list
  + Use max() and min() to return the highest and lowest numbers in the list respectively
* .index() returns the index position of a specific item in the list
* .count() returns the number of occurrences of some item within a list
* .join() compile your list items into a single string
* .split(<by\_char>) will split a string at each occurence of and group those items in a list
* .sort() or sorted() will sort a list in ascending order
  + Pass reverse = True to either one to sort the list in descending order

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## Loops

### **Iterating with Loops**

In programming, we define iteration to be the act of running the same block of code over and over again a certain number of times.For example, say you want to print out every item within a list. You could certainly do it this way -

|  |
| --- |
| visible\_colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]  print(visible\_colors[0]) print(visible\_colors[1]) print(visible\_colors[2]) print(visible\_colors[3]) print(visible\_colors[4]) print(visible\_colors[5]) print(visible\_colors[6]) |

Attempting to print each item in this list - while redundant - isn't so bad. But what if there were over 1000 items in that list? Or, worse still, what if that list changed based on user input (ie: *either* 10 items *or* 10000 items)?

To solve such problems, we can create a **loop** that will iterate through each item on our list and run the print() function. This way, we only have to write the print() one time to print out the whole list!

When you can iterate through an object (e.g. a string, list, dict, tuple, set, etc.), we call it an iterable object. Python has many built-in iterables. You can reference some of the most common ones in the itertools module (read more about itertools [here](https://realpython.com/python-itertools/)). You can also define your own Python iterables using the principles of OOP (object-oriented programming). In fact, Python features a more advanced construct called a generator to simplify this process for you.

For now, let's print each color in the visible spectrum again, but take one step toward making this more dynamic. We'll use a variable called idx to represent the index positions in visible\_colors. We'll set idx = 0 to start, since the first list element is at index 0.

|  |
| --- |
| visible\_colors = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]  idx = 0  print(visible\_colors[idx]) # idx = 0 idx = idx + 1  print(visible\_colors[idx]) # idx = 1 idx = idx + 1  print(visible\_colors[idx]) # idx = 2 idx = idx + 1  print(visible\_colors[idx]) # idx = 3 idx = idx + 1  print(visible\_colors[idx]) # idx = 4  idx = idx + 1  print(visible\_colors[idx]) # idx = 5 idx = idx + 1  print(visible\_colors[idx]) # idx = 6 |

As you can see, you simply have to increase the value of idx by 1 after each item you print. That way, we can repeat print(visible\_colors[idx]) to access ALL elements in the list without typing in their unique index numbers. If you consider this code conceptually...

### **the while loop**

...it is functionally the same as the while loop below.

|  |
| --- |
| idx = 0  **while** idx < len(visible\_colors):  print(visible\_colors[idx])  idx = idx + 1 |

The loop represents an abstraction of the code above, wherein we use a boolean expression to determine how many times to run the code in the loop's body.

At each iteration, we increase the value of idx by 1. As long as idx < len(visible\_colors) evaluates to True, the loop keeps running. Once idx < len(visible\_colors) evaluates to False, the loop ends and your program moves onto the next code block.

#### **Counting**

Here's another very basic use case for the while loop:

|  |
| --- |
| i = 0 **while** i < 10:  print(i)  i += 1 print(i) # will print out numbers 1 through 10 |

What is happening here is we are running the code block within the while 10 times. We know to stop because the boolean comparison will evaluate to False once i exceeds 10, which is possible only because i is being incremented when we write i += 1.

Use a loop to make a list of all odd numbers between 1 and 10 and another list with all the evens in this range. When done, print the lists.

|  |
| --- |
| odds = [] evens = []  i = 1  **while** i <= 10:  **if** i % 2 != 0:  odds.append(i)  **else**:  evens.append(i)  i += 1  print(odds,'\n', evens) |

#### **Booleans**

While loops are based on boolean expressions. They effectively direct the program to "do this action as long as this condition is True." Here's real-life scenario where you might apply a while loop. Let's say you've programmed your Amazon Echo or Google Home to make a pot of coffee whenever you say the trigger word "tired". Once you say tired, here's a *simplified pseudo-code version* of what happens behind the scenes:

|  |
| --- |
| tired = **True** **while** tired:  print('I\'ll make some coffee!') # this might be a "say" command  # code to turn on coffee maker  tired = **False** |

Whenever a pot of coffee is made, the smart device sets tired back to False. Next time you say "tired", it will reset tired to True.

### **the for loop**

Let's go back to that list of colors we wanted to print out and use a for loop. The most important part of the for loop is the statement for item in obj. This means the code **considers each item in the iterable one at a time** when executing the code below.

|  |
| --- |
| # Syntax: # for <item> in <iterable>: # <statement(s)>   visible\_colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'] **for** color **in** visible\_colors:  print(color) |

### **Loops with Ranges**

If you want to iterate through only a section of a list, the range() and enumerate() functions can facilitate this. range and enumerate are also both their own data type objects.

#### **range()**

With while loops, we saw one way to iterate while counting. Using range() with a for loop allows us to be more concise and more specific.

The range() function uses this syntax: range(<begin>, <end>, <stride>). It returns a range iterable that yields values starting with the index, *up to but* ***NOT*** *including the index*. The argument isn't required, but if specified, it indicates an amount to skip between values. For example, range(5, 20, 3) would iterate through 5, 8, 11, 14, and 17. If is omitted, it defaults to incrementing by 1.

Consider the differences in the loops below...

##### **range() with a while loop**

|  |
| --- |
| visible\_colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'] print(len(visible\_colors), '\n')  idx = 3  **while** 2 < idx < 6:  print(visible\_colors[i]) # prints green blue indigo  idx += 1 |

##### **range() with a for loop**

|  |
| --- |
| visible\_colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet'] print(len(visible\_colors), '\n')  x = range(3, 6)  **for** i **in** x:  print(visible\_colors[i]) |

#### **enumerate()**

When looping through a list, you might want to keep track of the index position of the item in each loop iteration. Converting the list it into an enumerate object allows you to achieve this more easily.

Take a look at what an enumerate object looks like below. Note that you can print the object as a whole to view its contents like you do with a list.

|  |
| --- |
| visible\_colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet'] enum = enumerate(visible\_colors)  print('Print whole list...', visible\_colors) print('Print whole enumerate object...', enum) |

You have to loop through them and print each one. As you do this, you'll see that each item in the enumerate object is a tuple.

|  |
| --- |
| visible\_colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet'] visible\_colors = enumerate(visible\_colors)  **for** i **in** visible\_colors:  print(i) |

But you can access each item in each tuple like this...

|  |
| --- |
| visible\_colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet'] visible\_colors = enumerate(visible\_colors)  **for** idx, i **in** visible\_colors:  print(f'{idx} & {i}') |

### **Control Flow with break, continue**

Something very important to watch out for here is falling into an **infinite loop**. This is one of the most common traps and can make your code go crazy running the loop over and over without moving through the rest of the program! It happens when you do not have proper control flow in the loop's code.

The **break keyword**, and the **continue keyword** are core ways to help control the flow and logic within your loops.

#### **The break Keyword**

In a Python loop, the break keyword *escapes the loop*, regardless of the iteration number and regardless of how much of the loop code it has completed on its current iteration. Once a break executes, the program will continue to execute after the loop.

We might use a break statement if we only want the loop to iterate under a certain condition. For example:

|  |
| --- |
| a = ['foo', 'bar', 'baz', 'qux', 'corge'] **while** a:  **if** len(a) < 3:  **break**  print(a.pop()) print('Done.')  ## This loop will output... """ corge qux baz Done. """ |

Let's walk through the logic of how we got that outcome:

|  |
| --- |
| a = ['foo', 'bar', 'baz', 'qux', 'corge'] **while** a: |

* ^^^ This tells us that as long as a is True - essentially, as long as it exists - go ahead with the next loop iteration.

|  |
| --- |
| **if** len(a) < 3:  **break**  print(a.pop()) |

* ^^^ This says that, if the length of a is less than 3, break out of the loop. In the first iteration, a has 5 items. Given this, the break is not executed. Instead, the code removes a random item from a and prints it. Once the loop gets to the 4th iteration, len(a) is 2. This triggers the break.

After that, the program goes to the next line of code after the break, in this case print('Done.').

This works the same with a for loop as in the example below. Can you think through why we get the outcome foo here?

|  |
| --- |
| **for** i **in** ['foo', 'bar', 'baz', 'qux']:  **if** 'b' **in** i:  **break**  print(i) # foo |

#### **The continue Keyword**

You can also use the continue keyword to interrupt the loop code. The difference is that the continue keyword escapes *only the current iteration*. A break escapes the loop entirely and goes on to execute the code immediately following the loop. A continue tells the program to stop where it is within the within the current iteration and *skip to the the next iteration* of the loop.

Here's an example using a while loop. Notice that the continue applies to the *outer* while loop, whereas the break applies only to the *inner* while loop.

|  |
| --- |
| # Prints out 0,1,2,3,4 s = ''  n = 5 **while** n > 0:  n -= 1  **if** (n % 2) == 0:  **continue**   a = ['foo', 'bar', 'baz']  **while** a:  s += str(n) + a.pop(0)  **if** len(a) < 2:  **break**  print(s) # '3foo3bar1foo1bar' |

As the program iterates through the decreasing values of n, it determines whether each value is even. The continue executes *only for these even-number iterations*. Then the loop continues to the next iteration. Thus, the inner while loop only initiates when n is 3 and 1.

Inside the inner while loop, a.pop(0) removes the first item of a. Once this has occurred twice, yielding 'foo' and 'bar', a has fewer than two items, and the break terminates the inner loop. Thus, the values concatenated onto s are, in turn, 3foo, 3bar, 1foo, and 1bar.

Again, this works the same with for loops like so:

|  |
| --- |
| **for** i **in** ['foo', 'bar', 'baz', 'qux']:  **if** 'b' **in** i:  **continue**  print(i) # foo, qux |

### **Infinite Loop Example**

See if you can figure out why this loop is infinite. (Tip: Don't actually run this code, or your computer will freak out!)

|  |
| --- |
| a = ['foo', 'bar', 'baz', 'qux', 'corge'] **while** a:  **if** len(a) < 3:  **continue**  print(a.pop()) print('Done.') |

Got it? After the first three iterations, a shrinks to fewer than three items and executes a continue statement. It then returns to the beginning of the loop, where it will find that a still has fewer than three items. So it goes back to the beginning again... and again and again and again...

Your program will get stuck here, so you want to make sure you pay special attention to the control flow when you write loops!

## 

## Dicts

### **Intro**

In addition to lists, another more comprehensive method for storing complex data are **dicts**, or dictionaries. In this lesson, you'll:

* Define the rules for the structure of dict objects
* Create dicts using 4 different methods
* Access dict data
* Edit dict content

### **Dict Structure**

In the example below, we associate a key (e.g. 'taq') to a value (e.g. 'karim'). Instead of being enclosed in [], dicts are enclosed in {}.

|  |
| --- |
| my\_dict = {  'key' : 'value' } |

The keys and values of a single dict don't have to be homogenous. In other words, you can mix and match different key, value, and key/value pair data types within one dict as seen below.

|  |
| --- |
| dict1 = {  'taq': 'karim',  'apple': 35,  **False**: 87.96,  35: 'dog',  'tree': **True**  # etc. }  print(dict1) |

The values in a dict can be any valid Python data type, but there are some restrictions on what you can use as keys:

1. *Keys* ***CAN*** *be strings, integers, floats, booleans, and tuples.*
2. *Keys* ***CANNOT*** *be lists or dicts.*

Do you see the pattern here so far? The data in a *dict key must be immutable.* Since lists and dicts are mutable, they cannot be used as keys in a dict. That said, they *CAN* serve as the values in a dict.

|  |
| --- |
| dict2 = {  47: [12.1, 'blue', **True**], # list as a dict value  'julianna': {**False**: 'cat'} # dict as a dict value } |

* *Also, the keys in a dict* ***must be unique****.* You'll see why shortly... But remember -- be careful not to add a key to a dict a second time. If you do, the second item will ***override*** the first item.

### **Creating Dicts**

There are several ways you can create your dict, but we'll go through the most basic ones here.

#### **Method 1: Pass in key value pairs directly using {}:**

|  |
| --- |
| food\_groups = {} # this creates a new, empty dict  food\_groups = {  'pomegranate': 'fruit',  'asparagus': 'vegetable',  'goat cheese': 'dairy',  'walnut': 'legume' }  print(food\_groups) |

#### **Method 2: Create an empty dict with the dict() method.**

Just like with the list() method, dict() only accepts one argument, meaning you have to pass in key/value pairs within the {}.

|  |
| --- |
| food\_groups = dict() # this creates a new, empty dict  food\_groups = dict({  'pomegranate': 'fruit',  'asparagus': 'vegetable',  'goat cheese': 'dairy',  'walnut': 'legume' })  print(food\_groups) |

#### **Method 3: Convert a *list of tuples* into a dict using dict()**

This is a good example of using typecasting on more complex data structures.

|  |
| --- |
| # list of tuples  listofTuples = [("Hello" , 7), ("hi" , 10), ("there" , 45),("at" , 23),("this" , 77)]  wordFrequency = dict(listofTuples) print(wordFrequency) # {'this': 77, 'there': 45, 'hi': 10, 'at': 23, 'Hello': 7} |

#### **Method 4: Use zip() to convert two lists into dict keys and values**

The zip() method takes the names of each list as parameters - the first list will become the dict's keys, and the second list will become the dict's values.

**NOTE!** *This only works if you're sure the key/value pairs have the same index position in their original lists (so they will match in the dict).*

|  |
| --- |
| names = ['Taq', 'Valerie', 'Viktor', 'Zola'] scores = [[98, 89, 92, 94], [86, 45, 98, 100], [100, 100, 100, 100], [76, 79, 80, 82]]  grades = dict(zip(names, scores)) print(grades) # {'Taq': [98, 89, 92, 94], 'Zola': [86, 45, 98, 100], 'Valerie': [76, 79, 80, 82]} |

### Accessing Dict Data

Once you've stored data in your dict, you'll need to be able to get back in and access it! Take a look at this dict holding state capitals.

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston' } |

#### **Referencing Values by Keys**

You *CANNOT* access dict items with index positions like you do with lists! If you try, you'll get a KeyError because dict items do not have index positions. **Instead, the dict keys serve the same purpose as indeces in lists.** Accordingly, you can access each value in the list by referencing its key like so:

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston' }  MAcap = state\_capitals['MA'] print(f'The capital of MA is {MAcap}.') # 'The capital of MA is Boston.' |

Attempting to find a key that does not exist leads to error.

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston' }  print(state\_capitals['PA']) # KeyError from missing key print(state\_capitals[2]) # KeyError from index reference |

Instead, it's better to look up a key in a dict using .get(key). The .get(key) method takes the key argument just as above EXCEPT it allows you to enter some default value it should return if the key you enter does not exist. Usually, we use [] as that value so that it's .get(key, []).

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston' }  print(state\_capitals.get('PA', [])) # PA is not in our dict, so .get() returns [] |

#### **Retrieving All Keys, Values, & Key/Value Pairs**

Now, this dict has 4 keys, but what if it had *hundreds?* We can isolate pieces of the dict's data structure using these functions:

* .keys() -- returns a collection of all the keys in a dict
* .values() -- returns a collection of all the values in a dict
* .items() -- returns a collection of all the key/value pairs in a dict

##### **Isolating Keys & Values**

Let's separate the keys and values from the pets dict below.

|  |
| --- |
| pets = {  'Taq': ['teacup pig','cat','cat'],  'Francesca': ['llama','horse','dog'],  'Walter': ['ferret','iguana'],  'Caleb': ['dog','rabbit','parakeet'] }  pet\_keys = pets.keys() pet\_values = pets.values() |

You would think the .keys() and .values() functions return lists of the keys and values repsectively, right? Wrong. These functions return *list-LIKE* objects called dict\_keys() and dict\_values(). Run this cell to see the results summarized for you.

|  |
| --- |
| print(f''' Keys:  {pet\_keys} {type(pet\_keys)} ''')  print(f''' Values:  {pet\_values} {type(pet\_values)} ''') |

In contrast to lists, you CANNOT access the elements in either a dict\_keys or a dict\_values object by index. Here's what happens if you try to:

|  |
| --- |
| print(pet\_keys[0]) # TypeError: 'dict\_keys' object is not subscriptable |

The same error would occur if you tried that with a dict\_values object. Because of this, it's often best to convert the objects to lists when you create them.

|  |
| --- |
| pets = {  'Taq': ['teacup pig','cat','cat'],  'Francesca': ['llama','horse','dog'],  'Walter': ['ferret','iguana'],  'Caleb': ['dog','rabbit','parakeet'] }  pet\_keys = list(pets.keys()) pet\_values = list(pets.values()) |

Then you can easily access each key or value by index:

|  |
| --- |
| print(f'First Key: {pet\_keys[0]}') # 'Taq' print(f'First Value: {pet\_values[0]}') # ['teacup pig','cat','cat'] |

##### **Isolating Key/Value Pairs**

You can access the full group of key/value pairs with .items(). Accordingly, .items() one will return a dict\_items object.

|  |
| --- |
| pets = {  'Taq': ['teacup pig','cat','cat'],  'Francesca': ['llama','horse','dog'],  'Walter': ['ferret','iguana'],  'Caleb': ['dog','rabbit','parakeet'] }  pet\_kv\_pairs = pets.items()  print(f''' Key/Value Pairs:  {pet\_kv\_pairs} {type(pet\_kv\_pairs)} ''') |

It looks like a list of tuples, right? Again, you'd think you could access each pair's tuple by index then, but you can't without first converting the dict\_items object to a list like we did before.

|  |
| --- |
| pets = {  'Taq': ['teacup pig','cat','cat'],  'Francesca': ['llama','horse','dog'],  'Walter': ['ferret','iguana'],  'Caleb': ['dog','rabbit','parakeet'] }  pet\_kv\_pairs = list(pets.items()) print(f'Key/Value Pairs: \n{pet\_kv\_pairs}\n{type(pet\_kv\_pairs)}\n\n') # [('Taq', ['teacup pig','cat','cat']), ('Francesca', [['llama','horse','dog']), etc]  print(pet\_kv\_pairs[0]) # ('Taq', ['teacup pig','cat','cat']) |

We'll learn *WHY* this is useful in the next section on loops!

### Built-in Operators for Manipulating Dicts

Just like lists, you can edit, analyze, and format your dicts. Some work the same for dicts and lists such as len() -- that will give you the number of key/value pairs in the dict. However, adding, deleting, and updating data requires a little more detail for dicts than for lists.

#### **Add or Edit Dict Items**

You can add a single item to dict in two ways. The first way is similar to updating a list...

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston' }  state\_capitals['CA'] = 'Sacramento'  print(state\_capitals) # {'NY': 'Albany', 'NJ': 'Trenton', 'CT': 'Hartford', 'MA': 'Boston', 'CA': 'Sacramento'} |

...but more likely you'll want to use the .update(key, value) method.

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston' }  state\_capitals.update('CA': 'Sacramento')  print(state\_capitals) # {'NY': 'Albany', 'NJ': 'Trenton', 'CT': 'Hartford', 'MA': 'Boston', 'CA': 'Sacramento'} |

The .update() method also allows you to make bulk updates. In that case, you can simply pass it a variable containing another dict to add to the first one.

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Trenton',  'CT': 'Hartford',  'MA': 'Boston',  'CA': 'Sacramento' } more\_states = {  'WA': 'Olympia',  'OR': 'Salem',  'TX': 'Austin',  'NJ': 'Hoboken',  'AZ': 'Phoenix',  'GA': 'Atlanta' }  state\_capitals.update(more\_states)  state\_capitals = {  'NY': 'Albany',  'NJ': 'Hoboken',  'CT': 'Hartford',  'MA': 'Boston',  'CA': 'Sacramento',  'WA': 'Olympia',  'OR': 'Salem',  'TX': 'Austin',  'AZ': 'Phoenix',  'GA': 'Atlanta' } |

**Notice something?** It's easy to accidentally override items when you're merging datasets. *Oops, we just changed the capital of NJ to Hoboken!* Don't worry though - we'll learn an easy way to check for duplicate keys in the next section on loops.

#### **Remove Items from a Dict**

##### **.clear() simply empties the dict of all items.**

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Hoboken',  'CT': 'Hartford',  'MA': 'Boston',  'CA': 'Sacramento',  'WA': 'Olympia',  'OR': 'Salem',  'TX': 'Austin',  'AZ': 'Phoenix',  'GA': 'Atlanta' }  state\_capitals.clear() print(state\_capitals) # {} |

##### **.pop(key, value):**

This removes an item, which you must specify by key. There are two things to note here -

1. **First**, you *cannot delete a dict item by specifying a value*. Since values do not have to be unique the way keys are, trying to delete items by referencing values could cause issues.
2. **Second**, just like we saw earlier with .get(key, value), .pop(key, value) will raise a KeyError if you try to remove a key that does not exist in the dict. We avoid this in the same way, by setting a default value - typically [] - for the program to return in case of a missing key.

Unfortunately, you *can't* use the same method as we did for .update() to delete larger portions of data. We'll learn a way to do that in the next section on loops as well.

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Hoboken',  'CT': 'Hartford',  'MA': 'Boston',  'CA': 'Sacramento',  'WA': 'Olympia',  'OR': 'Salem',  'TX': 'Austin',  'AZ': 'Phoenix',  'GA': 'Atlanta' }  state\_capitals.pop('AZ', []) # removes 'AZ': 'Phoenix' from our dict |

##### **popitem():**

This one just removes an arbitrary key value pair from dict and returns it as a tuple. For instance, you might do this if you're randomly sampling your data for QA purposes.

|  |
| --- |
| state\_capitals = {  'NY': 'Albany',  'NJ': 'Hoboken',  'CT': 'Hartford',  'MA': 'Boston',  'CA': 'Sacramento',  'WA': 'Olympia',  'OR': 'Salem',  'TX': 'Austin',  'AZ': 'Phoenix',  'GA': 'Atlanta' }  seceded1 = state\_capitals.popitem() # ^ removes a random item and returns it as a tuple print(seceded1) # ('GA': 'Atlanta') for example |

### **Iterating Through Dicts**

Iterating over dicts is slightly more complicated than other iterabless because each item consists of two elements, specifically mapped to each other. That said, you can do some really cool stuff with your dicts using loops!

##### **Iterate Through Dict Items**

Let's start with a few simple examples. This first one iterates over the dict by each *item*, i.e. each key-value pair.

|  |
| --- |
| transaction = {  "amount": 10.00,  "payee": "Joe Bloggs",  "account": 1234 }  **for** key, value **in** transaction.items():  print(f'{key}: {value}')  """Output: account: 1234 payee: Joe Bloggs amount: 10.0""" |

##### **Iterate Through Dict Keys**

If you only have a dict's keys, you can still iterate through the dict. Notice the loop below results in the same output as the one above iterating through items.

|  |
| --- |
| transaction = {  "amount": 10.00,  "payee": "Joe Bloggs",  "account": 1234 }  **for** key **in** transaction.keys():  print(f'{key}: {transaction[key]}')  """Output: account: 1234 payee: Joe Bloggs amount: 10.0""" |

#### **Sorting Dicts with Loops**

##### **By Key**

|  |
| --- |
| transaction = {  "amount": 10.00,  "payee": "Joe Bloggs",  "account": 1234 }  **for** key **in** sorted(transaction.keys()):  print(f'{key}: {transaction[key]}')   """Output: account: 1234 amount: 10.0 payee: Joe Bloggs""" |

##### **By the Values of Each Key**

Note that the dict itself will not be sorted by the first value in each item. Because the keys are the unique element of a dict, you can only sort dict values *within each key*.

|  |
| --- |
| dict1 ={   "L1":[87, 34, 56, 12],   "L2":[23, 00, 30, 10],   "L3":[1, 6, 2, 9],   "L4":[40, 34, 21, 67]  }  **for** k, v **in** dict1.items():   sorted\_pair = {k: sorted(v)} # here is sorting!  dict1.update(sorted\_pair)  print(dict1) """ # prints out... {'L1': [12, 34, 56, 87], 'L2': [0, 10, 23, 30], 'L3': [1, 2, 6, 9], 'L4': [21, 34, 40, 67] } """ |

### **Key Takeaways**

* To create a dict:
  + Pass comma-separated key : value pairs within {}
  + Pass a list of tuples to dict()
  + Pass two lists to zip()
* Unlike a list, a dict is inherently unordered and thus has no index.
* Access a value by referencing its key using my\_dict[key] or .get(key, []). The latter is preferable because it guards against KeyErrors from referencing missing keys.
* my\_dict[key] = value and .update(key, value) both add or update a specific key/value pair based on whether the key passed exists already in the dict or not
  + You can also pass another dict into .update() in order to make bulk edits
* .update(key, value) adds or updates a key/value pair based on whether the key passed exists already in the dict or not
* .pop(key, value) removes a key/value pair from the dict
* The following methods return list-like objects that isolate certains of a dict:
  + .keys() returns the keys
  + .values() returns the values
  + .items() returns the key/value pairs as tuples
* len(my\_dict.items()) will return the number of pairs in the dict
* To iterate through a dict, use one of the following as the context requires:
  + for key in my\_dict.keys():
  + for value in my\_dict.values():
  + for key, value in my\_dict.items():

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## Functions

A function is a reusable “container” that will abstract away the logic behind a task. For instance, suppose you are walking through rows of a CSV, converting each “line” which is a string delimited by commas:

**column\_name1, column\_name2**

**abc,123**

Into a dictionary:

**{“column\_name1”:”abc”, “column\_name2”:123}**

This is an action that almost certainly ought to be written as a function. A function allows the programmer to define “inputs” that are then transferred into deterministic “outputs” regardless of use case. For instance, in this example, it doesn’t matter if our CSV file describes temperature fluctuations in NYC over 4 years or if it describes CitiBike usages in manhattan this past month. Regardless of the *context*, if a string (input) that is delimited by commas is passed into it, it would output a dictionary (output) that maps the column names to column value for a single row.

### **Anatomy of a Function**

For now, let's start with the basics. Here's the skeleton of a function and a breakdown of each part.

|  |
| --- |
| **def** **function\_name**(parameters):  """docstring"""  # statement(s) |

#### **Naming & Input**

* def shows you are "defining" a new function
* A unique function\_name (same naming rules as variables)
* Optional variables, or parameters, that represent what input the function needs in order to run correctly
* : ends the function header

#### **Function Body**

* An optional docstring, i.e. a comment that documents the purpose and workings of the function
* At least one statement is needed in the function body; this code achieves the purpose for calling the function.

#### **Output**

* An optional return statement, which exits the function and passes out some return value(s) from the function's body code.

### **A Simple First Function**

Let's say we want to create a function to get the square of a number. At the most basic level, there are three parts:

1. Input the number we want to square
2. Calculate the square of that number
3. Output the square of that number

Let's implement this in a function called num\_squared.

|  |
| --- |
| **def** **num\_squared**(num):  """Find the square of some number passed in"""  square = num\*num # code to find the square  **return** square |

#### **Input the number we want to square**

We create a parameter called num to represent the number we will pass into our function as an argument. Remember that a function can define multiple parameters, each uniquely named. When the function is “run”, or invoked, values must be provided for each parameter in the correct position - otherwise the function will not recognize them.

#### **Calculate the square of that number**

Using the value of num, we write the formula for calculating a square and assign the result to the variable square.

#### **Output the square of that number**

We return square to pass on the numeric value we calculated. The return statement exits the function so the program can move on to the next block of code written. If a return statement is not specified, the function will by default return None once the lsat line in the function block has been run.

### **Types of Function Input**

The term parameter and the term argument are often used synonymously to refer to function input values. However, there IS a distinction between the two terms:

* Parameters are values you declare when defining a function.
* Arguments refer to the values passed into the function upon calling it

### **Required vs. Optional Parameters**

You have a fair amount of flexibility when it comes to defining function parameters. This is because there are so many different use cases for the amount and type of input needed. The differences between function inputs are signified in the syntax of the function definition. We'll go through examples for the main use cases now...

keyword argument: an argument preceded by an identifier (e.g. name=) in a function call or passed as a value in a dictionary preceded by \*\*. For example, 3 and 5 are both keyword arguments in the following calls to complex():

complex(real=3, imag=5) complex(*\*{'real': 3, 'imag': 5}) positional argument: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by* . For example, 3 and 5 are both positional arguments in the following calls: complex(3, 5) complex(\*(3, 5))

**Required Parameters**

Below is how you'd define parameters for two *required* inputs with *no default values*. The variables For example...

|  |
| --- |
| **def** **plus**(a, b):  **return** a + b c = plus(8, 12) print(c) # 20 |

Whenever you call this function, you must pass values for each required argument *in the same positional order as they were defined*.

Keywords Arguments If you want to make sure that you call all the required arguments in the right order, you can use the **keyword arguments** in your function call. Essentially, this means that you mention each argument's parameter name when you assign it a value during the function call. It works like this...

|  |
| --- |
| **def** **plus**(a, b):  **return** a + b c = plus(a=8, b=12) print(c) # 20 |

#### **Default Input Values**

If you want, you can give your parameter a **default argument**. Doing so means that you don't have to explicitly pass a value for that argument when you call the function. In that case, the function assumes that you've implicitly passed the pre-defined default value to that argument.

Here's a version of our plus() function with default arguments for b and c:

|  |
| --- |
| **def** **plus**(a, b = 12, c = -1):  **return** a + b # Explicitly passing values for b & c sum1 = plus(8, b = 4, c = 3) print(sum1) # 15  # Implicitly passing default values for b & c sum2 = plus(8) print(sum2) # 20 |

Here's where things get a little sticky between how terms are colloquially used and how they actually work in Python...

1. In practice, b and c are called **keyword arguments** because you often reference their keyword when you do want to pass them some explicit value.

|  |
| --- |
| # Explicitly passing values for b & c sum3 = plus(8, b = -2, c = 6) print(sum3) # 12  # Implicitly passing default values for b & c sum4 = plus(8) print(sum4) # 20 |

That said, you CAN certainly reference the keyword for required arguments as well. It even makes sense to do so when you're using complex built-in functions. For example, some pandas functions have several required parameters AND more than a few optional ones.

|  |
| --- |
| # ...vs. passing values for `a` and implicitly `b` y = plus(a = 4, c = -1) print(y) # 15  # ...vs. passing values for `a` and explicitly `c` z = plus(8, c=12) print(z) |

**NOTE!** Technically speaking, these parameters are still required. However, think about it what happens if you give a parameter a default value of None and then don't specifically pass in a value when you call the function. *Effectively*, that parameter is optional.

Parameters with default arguments take some pre-defined default value if no argument value is passed when you call the function. In practice, such parameters are called **keyword arguments** because you

If you want to make sure that you call all the required arguments in the right order, you can use the **keyword arguments** in your function call. Essentially, this means that you mention each argument's parameter name when you assign it a value during the function call. It works like this...

Unlike required arguments, keyword arguments can be passed to the function in any order. Moreover, since you can assign None as a default value, keyword arguments are effectively optional. As a best practice for when you define your functions and when you use functions, you should place the required arguments first, then the optional ones.

|  |
| --- |
| **def** **full\_name**(first, last, middle = None, prefix=None, suffix = None):  parts = [prefix, first, middle, last, suffix]  name = ''  **for** i **in** parts:  **if** i **is** **not** **None** **and** len(name) != 0:  name += f' {i}'  **elif** i **is** **not** **None**:  name += f'{i}'  **return** name  name1 = full\_name('Taq', 'Karim') name2 = full\_name('Julianna', 'Garreffa', prefix = 'Ms.') name3 = full\_name('Rupert', 'Buckworth', suffix = 'III', middle = 'Malcolm', prefix = 'Mr.')  print(name1) print(name2) print(name3) |

(For simplicity's sake, let's assume all the inputs for this function are strings!)

#### **Variable number of Arguments**

Even if you're not sure how many arguments you will need to pass to your function, you can still define it. To do this, you use the parameter \*args as a stand-in. This signals to the function that it should expect any variety of arguments. Let's take a look at a few different ways to implement this.

**Using integers** (as we did in the earlier examples)

|  |
| --- |
| **def** **plus**(\*args):  **return** sum(args)  c = plus(8,12,17) print(c) # 37 |

**Using different data types**

|  |
| --- |
| **def** **length**(\*args):  list1 = [\*args]  **return** len(list1)  c = length(8,'a',**True**) print(c) # 3 |

**Using a variable**

|  |
| --- |
| var1 = 'h' + 'i' **def** **print\_all**(\*args):  list1 = [\*args]  **return** list1  c = print\_all(8,'a',**True**,var1) print(c) # [8, 'a', True, 'hi'] |

**NOTE!** If you use \*args, your function will be more flexible, *but only if you write it that way*. If you expect different types of arguments, you will have to write the function such that it can handle every use case you expect could occur.

### **Variable Scope Recap**

* global variable: a variable declared outside a function; any function in your script can access this
* local variable: a variable declared within a function's code block; you can only access this variable within the function where it is declard, otherwise you will get a NameError telling you that variable is not defined.

|  |
| --- |
| x = 'I\'m a global variable.'  **def** **foo**():  x = 'I\'m a local variable.'  print(x) # I'm a local variable.  **return** x  y = foo()  print(x) # I'm a global variable. print(y) # I'm a local variable. |

Notice that even though the function foo() above says return x, it **only returns the value of the local variable x**. We assign this *value* to the variable y when we call foo().

Look at the nuanced difference in this example though:

|  |
| --- |
| **def** **foo**():  x = 'I\'m a local variable.'  print(x) # I'm a local variable.  **return** x  foo()  print(x) # NameError: name 'x' is not defined |

Even though we called the function foo(), we did not assign its return value to a variable outside the function. Therefore, trying to print x will output NameError: name 'x' is not defined. This is because x only exists within the function.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## Classes & Inheritance

We already know that Python is based on the concept of **OOP, or Object-Oriented Programming**. Almost everything in Python is an object -- even functions are objects. **Classes**, and their facilitation of **inheritance**, are one of the most important and valuable Python objects. In this section, we'll cover:

* Class structure
* Class attributes
* Class methods
* The \_\_init\_\_() method
* The self keyword
* Class vs. instance variables
* Class instantiation
* Inheritance and child classes

### **High-Level Overview**

#### **Creating & Structuring Classes**

A **class** is essentially a data structure that serves as a blueprint for categorizing other objects and storing metadata about them. Once you have your "blueprint", you can create new **instances** of that class, which store unique metadata values.

Creating a class is similar to defining a function. You start with the class keyword and then specify a name for the class. Note that class names are generally the only objects, which use a **CamelCase notation** naming convention. For example, if you were a zoologist, you might create a class called Animal. Each instance might represent a type of animal at your zoo.

|  |
| --- |
| # Define a class called Animal **class** **Animal**:  # attributes  # methods  # etc ...  # Create the most basic instance chameleons = Animal() |

Before we go into the details of thoroughly defining a class, let's isolate some basic elements and concepts to get a general understanding of them.

#### **Attributes & Methods**

Each piece of a class's metadata is called an **attribute**. Once you have your "blueprint", you can create new **instances** of that class, which stores unique attribute values. As a zoologist, you would want define your Animals class so that it could store attributes of each type animal at your zoo such as species, natural habitat, etc..

|  |
| --- |
| **class** **Animal**:  kingdom = 'Animalia' # attribute   # some other code... |

In addition to attributes, classes also contain custom **methods**. Methods are essentially functions that belong to the class. You can call a function without referencing any other object, but to call a method, you need to reference its class. *Thus, all methods are functions, but not all functions are methods.* We've already used some List methods like my\_list.pop(), my\_list.append(), my.list.insert(index), etc.. When you create a class, you can define methods to serve as shortcuts for actions you might want to call frequently on instances of your class.

|  |
| --- |
| **class** **Animal**:  # some other code...   **def** **method1**(self): # method  # some action |

Once you've defined attributes and methods, here's how you call them on your class instance:

|  |
| --- |
| chameleons = Animal() # Create the instance.  print(chameleons.kingdom) # 'Animalia'  chameleons.method1() # This completes the defined method operations. |

#### **Inheritance Basics**

Classes can inherit attributes and methods from other classes according to a parent-child class hierarchy. Naturally, a **child class** inherits from a **parent class**. When you define a brand new class, Python 3 implicitly uses the generic, built-in object as the parent class. That means, whether we explicitly see it or not, every parent class is also the child class of its own parent class!

In the context of our zoo example, the different instances of Animal each store general information about a certain type of animal. Imagine you want to expand on an instance of Animal called elephants. In order to document information about each elephant at the zoo, you might create an Elephant class that inherits from your Animal class. To do so, you use this general syntax:

|  |
| --- |
| **class** **Elephant**(Animal):  # attributes  # methods  # etc ... |

Although the child class has access to everything defined for its parent class, the child class can also override or extend the parent class's traits and behavior. Note that this *does NOT redefine the parent class*. The new attributes and methods the child class declares apply only to instances of the child class. Parent class instances still adhere to the original parent class specs. For example:

|  |
| --- |
| **class** **Animal**:  category = 'Animals'  # etc ...  **class** **Toucan**(Animal):  category = 'Birds'  # etc ... |

If you wanted, the Toucan class could simply inherit the category class attribute from its parent class Animal. In this case, every instance of Toucan would have the same value for category -- Animals. However, it makes sense that you'd want to differentiate further for the child class Toucan. To do that, you'd simply override category when you define Toucan by setting its value to Birds.

#### **The init() Method & the self Keyword**

When you create a new instance of your Class, you might want to it to exist in some default state. For example, you might want to initially assign default values for its attributes. In Python terms - *when you* ***instantiate*** *a new instance object, you* ***initialize*** *it with pre-defined default values*.

**The init() method** is where you give instructions for how you want each instance to exist in its initial state. Every time you instantiate a new instance object of your Class, you automatically invoke the \_\_init\_\_() method. That means when you create a new Class, the first thing you want to do is create its \_\_init\_\_() method. In general, the syntax looks like this:

|  |
| --- |
| **class** **Animal**():  **def** **\_\_init\_\_**(self):  # ... |

Notice we used the same notation as we did for defining functions. The \_\_init\_\_() method must have at least one argument, including the **self variable**. The self variable serves as a reference to the *current instance of the class*, and it must be the first parameter of *any* method in a class, including the \_\_init\_\_() method.

### **Class vs. Instance Variables**

Now we can get to the good stuff! As you define attributes and methods for your class, keep in mind their *scope*. If you want a certain attribute or method to be shared by ALL instances of a class, define it as a **class variable**. If you instead want it to be unique to each instance, define it as an **instance variable**. Before we see this in context, we first have to understand the two most basic elements of every Python class...

#### **The init() Method & the self Keyword**

When you create a new instance of your Class, you might want to it to exist in some default state. For example, you might want to initially assign default values for its attributes. In Python terms - *when you* ***instantiate*** *a new instance object, you* ***initialize*** *it with pre-defined default values*.

**The init() method** is where you give instructions for how you want each instance to exist in its initial state. Every time you instantiate a new instance object of your Class, you automatically invoke the \_\_init\_\_() method. That means when you create a new Class, the first thing you want to do is create its \_\_init\_\_() method. In general, the syntax looks like this:

|  |
| --- |
| **class** **Animal**():  **def** **\_\_init\_\_**(self):  # ... |

Notice we used the same notation as we did for defining functions. The \_\_init\_\_() method must have at least one argument, including the **self variable**. The self variable serves as a reference to the *current instance of the class*, and it must be the first parameter of *any* method in a class, including the \_\_init\_\_() method.

**NOTE!** Any methods defined inside the \_\_init\_\_() method **will NOT be *called*** upon instantiation.

### **Class Definition Example 1 - Basic Elements in Context**

Now that we've isolated each key component of classes, let's put everything together by completing the code for our zoology scenario. At the highest level, we define a class called Animal. The annotated code below illustrates how each key structural element we covered above fits into this task.

|  |
| --- |
| **class** **Animal**: # A.  **def** **\_\_init\_\_**(self, species = '', diet= ''): # B.   self.species = species # C.  self.diet = diet # C.   kingdom = 'Animalia' # D.   **def** **my\_kingdom**(self):  print(self.kingdom)   **def** **feed\_me**(self): # E.  **if** self.diet == 'omnivore':  food = 'plants and meat'  **elif** self.diet == 'carnivore':  food = 'meat'  **elif** self.diet == 'herbivore':  food = 'plants'  print(f'{self.species} eat {food}!')  **return** **None** |

* A. Animal is a *child class* of object as well as a potential *parent class*.
* B. Every time we instantiate a new class object, the \_\_init\_\_() method will automatically be called to initialize the instance's values.
* C. Each instance of the Animal class will store unique values for the **instance attributes** species and diet. By default these will be blank or Nonetypes, but each instance can have its own unique values for them.
* D. ALL instances of the Animal class will have the kingdom **class attribute** with the value Animalia.
* E. We can call **instance methods** my\_kingdom and feed\_me on ANY instance of the Animal class. **Note!** In my\_kingdom, we access the class variable kingdom, but still reference it using self.

### **Class Definition Example 2 - Child Classes & Inheritance**

Let's go into some more detail with a new child class for Animal. In the Elephant class below, we define \_\_init\_\_() method and its parameters, class attributes, and instance methods with the same syntax used for any class we might create. There are a few key differences annotated in the comments below.

|  |
| --- |
| **class** **Elephant**(Animal): # A.  **def** **\_\_init\_\_**(self, name, genus = '', species = '', habitat = '', age = None): # B.  self.name = name  self.genus = genus  self.species = species  self.habitat = habitat  self.age = age  self.taxonomy = {'Kingdom': Animal.kingdom, 'Class': self.common\_taxonomy['Class'], 'Family': self.common\_taxonomy['Family'], 'Genus': self.genus, 'Species': self.species} # C.   diet = 'Herbivore' # D.   common\_taxonomy = {  'Class': 'Mammalia',  'Family': 'Elephantidae',  }   **def** **summary**(self):  print(f'All about {self.name} -')  print(f'Elephant, age {self.age}\nHabitat: {self.habitat}\nDiet: {self.diet}\n\nTaxonomy:')  **for** k,v **in** self.taxonomy.items():  print(f'{k}: {v}') |

* A. Declares Elephant as a *child class* of Animal by adding Animal into it as a definition parameter.
* B. Notice that even though taxonomy is not a parameter for the \_\_init\_\_() method, we can still define it as an instance attribute upon every instantiation.
* C. If you look closely, you'll see that the values for taxonomy all come from different places.
  + Some of the taxonomy attributes are inherited from Animal; while
  + some are constant class attributes across all elephants; and
  + Others are instance attributes unique to each elephant at the zoo. This is a great opportunity to dissect the syntax for referencing attributes from different sources.
* D. Here's a potential "gotcha". Remember that the Animal class also had an attribute called diet? Elephant does NOT inherit the diet attribute's value from Animal. Why? Two reasons:
  + First, Elephant defines diet as a class attribute for itself. This would supersede any variable called diet from the parent class.
  + Second, for Animal, diet is an instance attribute. Even if Elephant didn't define any type of attribute called diet for itself, a child class *never inherits the instance attributes* from their parent.

#### **Class Instantiation & Modification**

Now we'll create the first instance of the Elephant class. To do so, you would pass arguments for the \_\_init\_\_() parameters defined above. This automatically invokes the \_\_init\_\_() method and assigns the values of the arguments you passed to your new instance attributes. Note that the name argument is required, but the rest are optional. Their values will default to empty strings if no argument for them is passed.

|  |
| --- |
| elephant1 = Elephant('Felicia', 'Elephas', 'Elephas maximus', '', 38) # Notice we passed the default empty string for the habitat argument.  You can access **or** modify any instance attribute like so: # Access print(elephant1.name) # Felicia   # Add value for an empty attribute print(elephant1.habitat) # empty string by default elephant1.habitat = 'Asian forests'   # Update an existing attribute value print(elephant1.age) # 38 elephant1.age = 39 # Update the value of the age attribute. print(elephant1.age) # 39  # Define a new instance attribute, which will apply only to elephant1. elephant1.weight\_pounds = 6000 |

Finally, here's what happens when we call the summary() instance method:

|  |
| --- |
| elephant1.summary()  # Here's the output """ All about Felicia - Elephant, age 38 Habitat: Asian forests Diet: Herbivore  Taxonomy: Kingdom: Animalia Class: Mammalia Family: Elephantidae Genus: Elephas Species: Elephas maximus """ |

#### **Checking Class Values**

In case someone who is not an expert zoologist like you needs to access the zoo's database of animals, that person could use the isinstance() function is used to determine if an instance is also an instance of a certain parent class. For this example, imagine you have already also defined another class called Toucan with the same input variables as our Elephant class.

|  |
| --- |
| # Is elephant1 an instance of Animal()? print(isinstance(elephant1, Animal)) # True  # Is toucan1 an instance of Elephant()? print(isinstance(toucan1, Elephant)) # False |

### **Key Takeaways**

* A **class** outlines a set of **attributes** and **methods**, which will help categorize other objects.
* To add objects to the class, you declare them as an **instance** of that class.
* **Class variables** store values belonging to ALL instances of a class, whereas **instance variables** store values unique to each instance.
* **The init() method** is where you give instructions for how you want each instance to exist in its initial state. Every time you instantiate a new instance object of your Class, you automatically invoke the \_\_init\_\_() method.
* The **self variable** serves as a reference to the current instance of the class, and it must be the first parameter of *any* method in a class, including the \_\_init\_\_() method.
* **Child classes** can inherit attributes and methods from **parent classes**.
* **Child classes** can also override parent attributes and behaviors without redefining the parent class.