Intermediate_Python_Participant_Copy

August 12, 2025

1 Intermediate Python

Welcome to the tutorial section of Intermediate Python! In this section, we will cover the following topics:

- Conditionals
- Loops
- Functions
- Classes, methods, and modules
- Working with advanced data structures, including dictionaries and JSON
- Reading, writing, and exporting text and JSON files

1.0.1 Conditionals

It's often the case that we want the behavior of our Python code to change based on whether or not some condition is met. In that case we would use conditional statements, often 'if-then' type of statements, e.g. 'if it's sunny out then walk the dog, otherwise, stay inside'.

In the case of Python, we start a conditional very naturally with the word **if**:

```
if x > 20:
  print('we are over capacity')
```

And we can extend this with **else**:

```
if answer == 'the answer':
  print('that's correct!')
else:
  print('try again')
```

We can extend this further still with **elif**:

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

A note on indentation: Typically, Python will ignore whitespace, though, not in the case of expected (or unexpected) indentation. Indentation is expected when the following line(s) of code are part of the code statment started on the previous line. We could technically keep everything

that's of a single statement on a single line, but that can get messy quick. Splitting our statement into separate lines with added indentation helps with readability, makes code easier to debug, and Python understands and expects it!

1.1 Logical Operators

Also known as 'Boolean operators', these are used to connect and modify logical statements. In Python we have **and**, **or**, and **not**.

When we use an 'and' operator to string together conditional statements, both statements need to evalute to true in order for the code to execute:

```
if len(list1) > len(list2) and len(list1) < 20:
  list1 = list1 + list2
```

With an 'or' operator, the code will execute as long as one of the statements is true:

```
if my_color == 'purple' or my_color == 'pink':
  color_list[0] = my_color
else:
  color_list = color_list + [my_color]
```

And finally, the 'not' operator flips the Boolean value of whatever the statement within it evaluates to:

```
if not(5 < x):
  print('x is out of the desired range')
```

Let's move on to some guided exercises:

[1]:
```
# Write a conditional statement that takes into account the height and width
# of a box, and only allows boxes shorter than 6 inches and thinner than 10␣
 ↪inches to pass.
```

[2]:
```
# Write a conditional statement that prints 'you may pass' for any string␣
 ↪except the following: 'forbidden' and ''
# For the two disallowed strings, print appropriate statements instead
```

Another quick note (comments): Comments are useful thoughout the process of writing code. Use comments to make notes for yourself, to exclude parts of code you don't want to run at the moment (e.g. when debugging), and to explain (to your future self or others) what sections of your code do.

1.2 Loops

The next thing we are going to cover today are loops. Loops are useful for when we have a chunk of code that we want to execute repeatedly, perhaps over a range of values or as long as some condition is met.

The two types of loops that you're likely to run into are **for** loops and **while** loops. The behavior of the two is similar though there are some slight differences in the logic and use cases of each.

When we want to iterate our code over a sequence of data values, be it characters in a string, a numerical sequence, or items in a list, we would use a for loop. The general form for the syntax of a for loop looks like:

```
for item in sequence:
  # some block of code
  # we want executed for
  # each item in our sequence
```

In the code above, 'sequence' can be either pre-defined or defined in-place, while 'item' is essentially an in-place variable that doesn't need to be defined elsewhere and iterates over the data objects in your sequence, e.g.:

```
canines = ['fox', 'coyote', 'wolf']

for animal in canines:
  print('a ' + animal + ' is a type of canine')
```

Let's give it a try!:

[15]:
```
# Iterate over the string stored in our variable below(i.e., a sequence of␣
 ↪characters!)
# print 'ping!' every time you encounter a 'p' and increment a count variable␣
 ↪each time

my_string = 'peter piper picked a peck of pickled peppers'
```

The other type of loop, a while loop, executes for as long as the condition holds true. In this case, be careful that you don't write a condition that's always true, or you'll get stuck in an infinite loop!

```
while count < 25 and snacks > 0:
  student_list += [new_student]
  print('welcome to today's session)
else:
  print('try again next week')
```

Example of an infinite loop:

```
while len(my_list) >= 0:
  print("i will run forever >:D\nunless you kill me... :'(")
```

If you're working interactively in a terminal, you can manually interrupt the execution of the loop by pressing ctrl+C (any OS). If the infinite loop is part of a script then you will need to have code written in to handle it. It's better to not end up with that situation in the first place if we can help it, so check your logic beforehand!

Let's try writing our own while loops below:

[16]:
```
# Write a loop that takes into account a temperature varaiable and prints the␣
 ↪statement 'the temperature is within the desired range'
# for a temperature between 70-75 degrees. To simulate a changing temperature␣
 ↪reading, increment the temperature variable by some
```

```
# amount each time the loop iterates
```

1.3 Functions

If instead we have some code or series of steps we may want to run multiple times, but not necessarily repeatedly as we've done with for and while loops, we can define a function to hold that code so that we're able to call it and run it whenever we like. And, unlike loops, we can define our function to take inputs (a.k.a parameters or arguments) which we pass into it when we call it. Functions can have as many parameters as needed, or none at all. Defining a function takes the following form:

```
def my_function(arg1, arg2):
  return arg1^2 + arg2^2
```

We use 'return' when we want our function to return a value. So with the example above if we had two numbers stored in variable x1 and x2 and wanted to store the result of running our function with those two variable as input into a new variable, x3, we would do:

```
x3 = my_function(x1, x2)
```

Parameters can be of any data type. When we pass an argument to a function it is treated as that input data type through the rest of the function's code. In the following example, we take a list as input and use the remainder function to create a new list with only values that sit at the input list's odd indices:

```
def list_function(my_list):

  new_list=[]

  for item in my_list:
    if my_list.index(item) % 2 == 1:

      print(my_list.index(item))

      new_list = new_list + [item]

  return new_list
```

A note on debugging: in the loop in the function above, the print statement isn't really necessary, but adding something like it to loops or conditional statements can help show if your code is running as expected or not.

How might we modify the above example to instead return a list of only the values at even indices of the input list?

Next, let's do a couple of guided exercises where we try writing our own functions:

[4]:
```
# write a function that takes a string as input and returns the length of it
```

[5]:
```
# write a function that takes a string and position as arguments and returns␣
↪the character at that position
```

1.4 Object Classes, methods, and properties

Python is an object-oriented programming language. This means that for all the built-in data types, or types of data objects we use, they have pre-defined rules for how they act and interact with the world.

These different data types are called Classes and generally each class can contain methods (built-in functions) and attributes or properties that can be called using the '.' method.

Some of our built-in data types in Python have some very useful methods that can be called, e.g.:

[1]:
```
list1 = [4,1,3,2]
# list1.sort()
# print(list1)

string1 = 'this is a string'
# string2 = string1.replace('i','j')
# print(string2)
```

Take a look at the following pages to see what other methods are available for strings and lists:

String methods: https://www.w3schools.com/python/python_ref_string.asp

List methods: https://www.w3schools.com/python/python_ref_list.asp

1.4.1 Defining Classes

We are also able to define our own classes! Somewhat similar to how defining a function allows you to use it whenever you need it throughout your script, creating an object class allows you to initialize new objects of that class whenever we need. As an example:

```
class rectangle:
 def __init__(self, length, width):
  self.length = length
  self.width = width
```

Important things to note are the '___init___' and 'self' portions in our defined class. '___init___' is an internal function that initializes class instances, or in other words, creates a new data object of that class type. And 'self' is a self-referential parameter that represents an object of that class. Once our class is defined, to create an instance of it we call the class name with whatever parameters it needs passed to it and store it in a variable:

```
my_rectangle = rectangle(10, 5)
```

Above we created a 'rectangle' class with a couple of attributes that we pass upon creation. To define a method within the class that can be called later we simply define a function within it:

```
class rectangle:
 def __init__(self, length, width):
  self.length = length
  self.width = width

 def area(self):
```

```
    area = self.length*self.width
    return area
```

[7]:
```
# define our class object
```

[8]:
```
# call an attribute of our newly defined class object:


# # call a method from our newly defined class object:
```

Further, classes can have subclasses and these can inherit all the methods and properties of it's parent class. Read more about classes and class inheritance in the links below:

https://www.w3schools.com/python/python_classes.asp

https://www.w3schools.com/python/python_inheritance.asp

Modules A module in Python is a file that contains Python code, which can include functions, classes, and variables. Modules allow for code organization and reuse, enabling you to import and use the functionalities defined in one module within another. They are incredibly useful, particular if you need to reuse certain functions in your code.

Modules are not the same as libraries.

Modules and libraries are often conflated. A group of modules can make up a Python library, but they are not the same thing.

In the example below, the module "my_module" is made up of two simple functions and stored in a .py file.

```
# my_module.py (This is the name of our module file).

def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b
```

To call a module, use the import statement. By importing the module into your Python environment, you can now access the functions you defined in the module.

```
import my_module

greeting = my_module.greet("Alice")
sum_result = my_module.add(5, 3)

print(greeting)
print(sum_result)
```

2 Advanced Data Structures in Python : Dictionaries and JSON

While working with Python, you may come across advanced data structures that have their own unique structures and methods. Working with these data structures requires additional care.

2.0.1 Dictionaries

Dictionaries in Python are structures used to store key:value pairs. Dictionaries in Python are ordered, changeable, and do not allow duplicate values.

Keys within a dictionary must be a string. However, values may be of any basic data type (string, integer, float, logical).

Note: Python versions 3.6 and earlier allowed for unordered dictionaries. Python versions 3.7 and later only allow ordered dictionaries. Double-check the version of Python you are using prior to working with dictionaries!

Defining a Dictionary A dicitonary in Python is denoted with curly brackets and key-value pairs. Separate each key:value pair with a comma. Indentation is recommended, but not required when defining a dictionary.

```
dictionary_1 = {
 "favorite_food" : "cheeseburger",
 "favorite_color" : "blue",
 "favorite_music" : "jazz"
}
```

Guided Exercise In the code box below, create a dictionary with three key value pairs: favorite food, favorite color, and your favorite music.

```
[ ]:
```

2.0.2 Accessing a Dictionary

To access a key within the dictionary, use square brackets and the name of the key you wish to access.

```
dictionary_1 = {
 "favorite_food": "pizza",
 "favorite_color": "pink",
 "favorite_animal": "cat"
}

food = dictionary_1["favorite_food"]
```

You can also use the **.get()** method to access a key in a dictionary.

```
food = dictionary_1.get("favorite_food")
```

To get a full list of the keys in the dictionary, use the **.keys()** method.

```
keys = dictionary_1.keys()
```

Guided Exercise In the code box below, use both methods of accessing keys within the dictionary you created earlier to access the key "favorite_color".

In addition, use the .keys() methods to obtain the keys of the dictionary "favorites."

[ ]:
```
```

Modifying a Dictionary There are several ways we can modify a dictionary within Python: * Adding key-value pairs * Removing key-value pairs * Updating the value of a key

Adding a key-value pair To add a key-value pair to your dictionary, include the name of the key you wish to add in square brackets. Assign a value using an equal sign.

```
my_dict["new_key"] = 50
```

Removing a key-value pair To remove a key-value pair from your dictionary, use the **.pop()** method. This will remove both the key and the value from the dictionary

```
my_dict.pop("key")
```

Updating the value of a key To update the value of a key, include the name of the key in square brackets. Assign a new value using an equal sign.

```
my_dict["existing key"] = 50
```

Guided Exercise In the code box below, please do the following: * Add the key "occupation" and the value "teacher" to the dictionary. * Remove the key-value pair "age" * Update the value of the key "city" to "Dallas".

[9]:
```
person = {
    "name": "John",
    "age": 50,
    "city": "Chicago"
}
```

Converting Between Dictionaries and Dataframes Python dictionaries can also be converted to pandas dataframes and back to dictionaries, which is incredibly useful if you regularly use the pandas library. However, there are several nuances that need to be accounted for in order to successfully convert between the two.

Before you continue... Use the code box below to import the pandas library into your environment.

[11]:
```
import pandas as pd
```

To convert a dictionary to a dataframe, use **pd.DataFrame()** and include the name of the dictionary in square brackets.

```
df = pd.DataFrame([dictionary])
```

Alternatively, you use the method **from_dict()** in conjunction with pd.DataFrame(). **from_dict()** requires an index in order to successfully convert, so if the values paired with the keys in your dictionary are singular, i.e. not part of a list, then you will need to pass either **orient = 'index'** as an extra option when running **from_dict()**, or go to your dictionary and make all your values as part of a list (single value lists are fine, what matters is that lists have an index for the conversion function to use).

```
df = pd.DataFrame.from_dict(dictionary, orient='index')
```

To convert from a dataframe to a dictionary, use the method **.to_dict().** To keep the content of your dictionary the same as the rows in the dictionary, include "records" in the parentheses. This tells Python to convert the dataframe by row.

```
df.to_dict("records")
```

Guided Exercise Use the methods above to convert a dictionary into a pandas dataframe. Take note of how the additional options (orient='index' and 'records') change the behavior of the conversion. Once you have accomplished that, convert the dataframe back to a dictionary with .to_dict(). What additional steps must be taken to get back to your original dictionary?

[ ]:

2.0.3 JSON

JSON (JavaScript Object Notation) is a string data format most commonly used for exchanging data between web clients and web servers. If you use Python to obtain data from an API or to send data to a database, knowing how to work with JSON in Python is essential. However, because of the differences in convention between JSON and Python, some care must be taken in order to successfully work with JSON in Python.

Importing the json Library To work with JSON in Python, import the **json** library. This library is included with base Python and does not require additional installation.

```
import json
```

Before you continue... Use the code box below to import the json library.

[31]:
```
import json
```

JSON Structure Similar to Python dictionaries, JSON stores key:value pairs and denoted in curly brackets. However, JSON has a few more restrictions than Python dictionaries. For example, key:value pairs in JSON *must* be denoted with double quotes.

Note: Python uses the data type None in place of null types. We will go over this more in detail in a moment.

```
{"name":"John", "age":30, "car":null}
```

JSON Strings vs. JSON Objects Occasionally, you will encounter the terms "JSON string" and "JSON objects," and often these terms are conflated. There are key functional differences between the two.

JSON strings are a string of characters, and can contain JSON objects.

```
'{"greeting": "Hello, World!", "language": "English"}'
```

JSON objects are structured collections of data denoted with curly brackets.

```
{
  "greeting": "Hello, World!",
  "language": "English"
}
```

In short, JSON strings can contain JSON objects, but JSON objects are not JSON strings. JSON objects will often appear with type "dict", indicating that Python is treating it as a dictionary.

Guided Exercise In the code box below, use the **type()** function to determine if the following are JSON strings or Python dictionaries/potential JSON objects:

```
'{"day" : "Monday", "course" : "Chemistry}'


{"day" : "Monday", "course" : "Chemistry}
```

[13]:
```
course1 = '{"day" : "Monday", "course" : null}'
course2 = {"day" : "Monday", "course" : "Chemistry"}
```

WARNING: Beware of "fake JSON!" If you're interacting with JSON using Python, it can turn into "fake JSON". That is, an acceptable Python dictionary, but not an acceptable JSON object, which will be an issue if you try to export it as is to an API that expects a properly formatted JSON object. Try to identify what's wrong with the "JSON" excerpt below.

```
{
 'source': {'id': 'https://openalex.org/S183686791',
  'display_name': 'The Astrophysical Journal Supplement Series',
  'issn_l': '0067-0049',
  'issn': ['0067-0049', '1538-4365'],
  'is_oa': True,
  'is_in_doaj': True,
  'is_core': None,
  'host_organization': 'https://openalex.org/P4310311669',
  'host_organization_name': 'Institute of Physics',
  'host_organization_lineage': ['https://openalex.org/P4310311669'],
  'host_organization_lineage_names': ['Institute of Physics'],
  'type': 'journal'}
}
```

Characteristics of "fake JSON" can include:

- Having null values appear as "None"
- Single quotes instead of double quotes

Converting between Dictionaries and JSON Objects

json.load() and json.loads() json.load() and json.loads() are used to convert JSON into workable Python objects. This is useful if you need to modify the JSON data or use it in conjunction with another library, such as pandas.

Note: The "s" matters! There is a significant functional difference between .load() and .loads(): * json.load() is used to parse a file containing a JSON object into a Python object * json.loads() is used to parse a JSON string into a Python object

If you're working with an API, you will use json.loads(), as this will convert a string received over a network into a Python object.

json.dump() and json.dumps()

Conversely, json.dump() and json.dumps() are used to convert Python objects into either a JSON file or string, respectively.

- json.dump() is used to write JSON data to a file. This method requires the use of several options in order to work properly, so we don't recommend it, though the interested reader can go to the following link to learn more: https://www.geeksforgeeks.org/python/json-dump-in-python/
- json.dumps() is used to convert Python objects into JSON strings for other purposes (printing, parsing, writing to a file, etc.). We recommend turning JSON objects into JSON strings and writing those strings to a text file to create JSON files instead.

You can also specify an indent within json.dumps() to get your JSON to "pretty print," or appear in a more readable format.

```
json.dumps(indent=3)
```

Guided Exercise Use json.dumps() to convert the dictionary flower_sample into a JSON string. Save the output to the variable "flower." Print the contents and type for variable "flower".

Once you have done that, convert the JSON string "flower" into a Python dictionary using json.loads(). Save the results of the conversion to the variable "rose". Print the contents and type for the variable "rose".

[14]:
```
flower_sample = {"flower" : "rose",
          "season" : "summer",
          "color" : "red"
         }
```

2.0.4 Reading, Writing, and Exporting Files

There are many cases while developing Python code you will need to read, write, and export different kinds of files in and out of your Python environment. This section will cover the basics of reading, writing, and exporting text and JSON files.

While this section will not cover all file types, much of the code can be adapted to work with other file types.

Text Files Reading and writing text files with Python is done in conjunction with the command **with open()**. This allows for easier file handling and memory usage, and helps ensure the file is closed if an error occurs. Alternatively, text files can be opened with the command **open()**, but this requires that you explicitly close the file after you're done with the command **close()**. Using **with open()** closes the file automatically immediately after.

Writing a Simple Text File To write a simple text file, first come up with the text you would like to include and save it into a variable.

```
text = "This is a sample text!"
```

Next, write a with open statement. Include the name of the file and file extension, as well as a "w". This signals that the file should be open in write mode.

Use an alias (nickname) for your file, such as "my_file," and end the line with a colon.

```
with open("sample_text.txt", "w") as file:
```

Finally, with the alias you just created, use the **.write()** method. Include the variable name containing the text in the parentheses, and make sure there is a single indent preceding it.

```
file.write(text)
```

Altogether, it looks like this:

```
text = "This is a sample text!"
with open("sample_text.txt", "w") as file:
  file.write(text)
```

When you run your code, it will create, write, save a text file, and close it.

Reading a Simple Text File To read a simple text file, follow same syntax as the with open for writing a file. Instead of a "w", include an "r". This signals that the file should be opened in read mode.

Create a variable to store your content, and use the **.read()** function to store the contents of your text file.

```
with open("sample_text.txt", "r") as file:
  content = file.read()
```

Guided Exercise Using the sample text below, write a simple .txt file. Once you have successfully saved your file, read the file contents into the variable "content."

[45]:

[ ]:

Reading, Writing, and Exporting JSON Files The process for reading and writing JSON files is similar to that of simple text files. However, there are a few addtional steps.

To write your JSON data to a file, ensure that your data is a JSON string by converting it with json.dumps().

```
file_data = json.dumps(data)
```

Similar to a text file, use a with open statement and include your desired file name and a "w". Ensure the file extension is .json to save it as a JSON file.

```
with open("sample_data.json", "w") as file:
  file.write(data)
```

To read a JSON file, follow the syntax for a with open statement for a text file. In the second line, use **json.load()** to parse the contents of the json file into a variable. This might be a little confusing since a JSON file is essentially just a text file, and we might equate text with strings, so we might think json.loads() is the right function to use, but when Python opens a file in read mode its not yet treated as a string until we .read() it.

```
with open('sample_data.json', "r") as file:
    data = json.load(file)
```

Guided Exercise In the code box below, write the data for flower_sample to a JSON file.Once you have successfully accomplished that, open the JSON file you just created and store the contents in the variable "loaded_JSON."

[17]:
```
flower_sample = {"flower" : "rose",
          "season" : "summer",
          "color" : "red"
         }
```

[ ]:
```

```

Conclusion & Additional Resources Thank you for attending Intermediate Python! Below are documentation and additional resources related to the topics discussed at this session.

PEP 8 Style Guide

https://peps.python.org/pep-0008/

json Library Documentation

https://docs.python.org/3/library/json.html

pandas Documentation

https://pandas.pydata.org/docs/

File Handling in Python (Geeks for Geeks)

https://www.geeksforgeeks.org/file-handling-python/

String Methods for Python

https://www.w3schools.com/python/python_ref_string.asp

List Methods for Python

https://www.w3schools.com/python/python_ref_list.asp

More info on json.dumps()

https://www.geeksforgeeks.org/python/json-dump-in-python/

[ ]: