Intro-to-Python_participant-copy

October 24, 2025

1 Introduction to Python

Hello, and welcome to Introduction to Python! In this tutorial, we will go over the basics of Python, including basic commands, data types and structures, and introducing additional modules that can be imported into Python.

1.1 Part 1: Syntax, Error Messages, and Writing your First Python Code

Approaching Python can be intimidating, especially to those new to programming. In this first part, we will go over the basics of Python syntax and some essential functions to get you started.

We'll also see a few error messages returned in the process. These are important for figuring out what went wrong with the code we're trying to execute.

1.1.1 Syntax

Syntax in the context of programming refers to the rules for punctuation, symbols, and words. In order for code to execute correctly, the syntax must be correct.

Writing Python code is similar to writing a "to-do" list for someone. The goal is to make sure everything is executed correctly, and to accomplish this goal, we need to make sure the "to-dos" are written in a way that is clear to the interpreter and able to be executed. Using proper grammar, spelling, and making sure the essential information is included in the code will allow Python to execute the prescribed task.

Here are some basic rules to get started: * Python is **case-sensitive**. If your cases don't match (ex. using an uppercase letter instead of lowercase), you will get a syntax error. * With the exception of certain data types (and also front whitespace), Python ignores whitespace. As in written English, using whitespace will make your code more readable. * Make sure your spelling is accurate. If you mistype the name of a variable or function, you will get a syntax error.

As you progress through the tutorial, more specific rules pertaining to syntax will be addressed.

[7]:

[ ]:

1.1.2 print()

Now that you have a general understanding of syntax, let's write some code!

An essential function in Python is the **print()** function. This will display the contents of whatever is in between the parentheses.

For example, if we wanted to use the print() function to say "Hello world!", it would look like this.

print("Hello world!")

The input is a **string**, a group of characters meant to represent text. Strings are contained within quotation marks in the context of functions. We will discuss strings in detail later in this tutorial.

Exercise 1: Hello world! In the code box below, use the **print()** function to display "Hello world!" Use either the Run button or shift+Enter to execute your code.

[1]:
```
# print 'hello world'
```

[ ]:
```

```

1.1.3 Operators

Operators are utilized to perform operations in Python. There are three types: arthimetic, logical, and assignment.

Arithmetic operators are used to perform mathematical operations:

- Addition : +
- Subtraction : -
- Multiplication : *
- Division : /
- Modulus : %
- Exponentiation: **

Assignment operators are used to assign values to variables. The most commonly used assignment operator in Python is =. Another useful one is the += operator. We will discuss how these operators will be used later in the tutorial.

Logical operators are meant to show comparison between two values. If the comparison is accurate, the output will be True. If the comparison is inaccurate, the output will be False.

- > : Greater than
- >=: Greater than or equal to
- < : Less than
- <= : Less than or equal to
- == : is exactly equal to
- != : is not equal to

Guided Exercise: Operators In the boxes below, we will experiment with the arithmetic and logical operators and examine how the outputs differ.

[ ]:
```

```

[ ]:
```

```

1.1.4 Error messages

Don't completely ignore error messages! They tell you what line of code Python broke at and give a short description of what went wrong! As an example, let's intentionally break Python by misspelling our variable name:

[2]:
```
# my_string = 'a bunch of text'
```

1.1.5 Comments

Comments describe and clarify code. They can also be used as part of a program's documentation, or as internal notes. They are intended for humans to read – not for a computer to translate and run.

In Python, comment lines begin with a single hash character: #. Any characters that follow and are part of the same line are will not execute as code.

Writing clear and useful comments makes code easier to read, maintain, adapt, and share.

Multiline Comments

Sometimes, you will want to comment out a block of code as opposed to a single line. In Jupyter Notebooks, multi-line comments are commented out using **ctrl + /** in linux and Windows systems and **cmd + /** on Mac while multiple lines of code are highlighted.

Guided Exercise: In the box below, use # to write a comment in Pyhon.

Bonus Exercise : Use **ctrl + /** (or **cmd + /**) to execute a multiline comment.

[7]:
```
# this bit of code does dfjklsajflk;jsakjfksjd

my_variable = 25 # this is my variable
# this piece of code does blah blah blah



# line of code
# line of code
# lines of code

# skfjksaljflkjskfjlksjljlsjfl
# sfksjfkjslkfjlasjflsklfjklasj
# fhsdfjklsajfkljsaklfjlds
# sflksajfkljsdljflksajfljs
# salfkjdskljfslfjlsjlf
```

1.2 Part 2: Data Types and Structures

Python is an object-oriented programming language, meaning that it is designed to work directly with data. On a more technical level it means that for all the built-in data types or data objects we use, they have pre-defined rules for how they act and interact with the world. In Part 2, we will go

over some basic data types and structures and some useful functions for converting and structuring data.

1.2.1 Data Types

There are four basic data types in Python. * **Strings**: Characters meant to represent text. * **Integers**: Whole numbers * **Floating point or "floats"**: Numerical values with a floating point decimal. * **Boolean**: Logical values indicated by either True or False.

It is common for "real-world" datasets to include values of all of these types.

1.2.2 Variables

Variables are containers for storing data values. Think of a variable as an empty box that can fit anything inside of it. The variable name is a label. Using an equal sign (=), we can assign values into variables.

Variable Names When naming your variables, it is essential that the name be meaningful and relevant. Someone looking at your code for the first time should be able to quickly ascertain what the values in each variable are meant to represent. Here are some general guidelines:

- Avoid generic names (x, y, num, etc.). This ambiguity can lead to confusion.
- Simple variable names are written in all lowercase letters. If more than one word is required, separate each word with an underscore or use camelCase or PascalCase.
- Variable names cannot start with a number, but can be used elsewhere in the variable name.

Good variable names include: * apples * total_fruit * my_variable2 * myCamels * TotalPets

Bad variable names include: * x * 2_bananas * onegg

[ ]:

Checking Variable Types If you're not sure what data type is contained within a variable, use the **type()** function. This is useful if you're working with a large number of variables or if the data types contained within the variables change often.

type(my_variable1)

1.2.3 Guided Exercise: Basic Data Types and Variables

In this exercise, we will experiment with the different basic data types and create variables. In addition, we will learn some basic functions for converting between different types.

[3]:
```
#Let's define some variables of different data types
# String:

# Integer:

# Float:

# Boolean:
```

[4]:
```
#use print to display variables
```

[5]:
```
# Try some operations on stuff that's not numerical



# What do comparison operators do with other data types?
```

[6]:
```
#convert between data types (note that non-sensical things won't work!):
```

1.2.4 Additional Data Structures: Lists, Tuples, and Dictionaries

Some times, you may want to store multiple items in one variable. To accomplish this, we can use a **list**, **tuple**, or **dictionary**.

Lists To create a list, use square brackets. Within the brackets, separate each item in the list with a comma. Items in the list can be any data type.

list1 = ['apples', 'bananas', 'grapes']

list2 = [1,2,3,4,5]

Indexing Lists Lists are dynamic and can be indexed.

IMPORTANT: When indexing lists, note that the index starts with zero (0, 1, 2, 3). This can be easy to forget!

list1[0], for example, would give the output 'apples', since this is the first item within the list.

Python also supports negative indexes. Think of this as "counting backwards": placing a negative in front of the index position tells Python to go back a defined number of spaces from the end of the output.

list1[-2] will give the output 'bananas', since this item is two entries from the end of the list.

[7]:
```
#indices
```

Modifying Lists and Useful Functions Lists can also be modified and appended. To accomplish this, you can use several methods:

- You can assign a new value to a specific entry in the list. For example, if we wanted to have peaches as the first entry of the list but not apples, you can assign the new value to the index position.

list1[0] = 'peaches'

- You can append values to the list using the addition operator(+). Please note that this will add the values to the end of the list.

list1 + ['oranges', 'avocados']

- To delete an item from the list, use the **del** command.

del list1[2]

- To get the length of a list, use the **len()** function. This is useful if you have a long list.

len(list1)

[8]:
```
# create a list, get items from it, and reassign an item
```

[9]:
```
#use the len() function, del command, and append to the list
```

Tuples Tuples are very similar lists in both structure and function, with two important distinctions:

- Tuples are defined using parentheses instead of square brackets.
- Immutable - tuples cannot be modified once they are created.
- If you're familiar with tuples in a math context as coordinates in a space, note that these are not that!

Tuples can be useful when you have data that does not need to be changed, such as an ordered set of keys, and the advantage gained from this over lists is in computational speed. For small applications this is unlikely to be significant, but if you're calculating over a huge list or doing very many computations than this speed difference begins to add up.

tup2 = ('apples', 'bananas', 'grapes')

[10]:
```
#tuples
```

1.2.5 Dictionaries

One last data type that we'll cover today is a dictionary. In Python a dictionary is a set of key:value pairs, where the key is like a variable name and it has a particular value attached to it. Each of the keys within a dictionary must be unique, and we use dictionaries to store structured information. Dictionaries in Python are ordered and changeable.

To represent a dictionary in Python we start with curly brackets {}, and within the brackets we put our key and value pairs separated from each other by a colon, and separated from other key:value pairs with commas. Keys within a dictionary are usually strings, though they can be numbers too. However, values may be of any data type (string, integer, logical, list, etc.)

Defining a dictionary: A dictionary can be defined all on one line like so:

```
my_dict = {key1:value1, key2:value2, key3:value3}
```

Or for easier reading, indentation is recommended:

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

Note that if you're using Python version 3.7 or above, dictionaries are ordered, meaning they will always display in the same order. In earlier versions of Python, dictionaries are unordered.

Accessing a Dictionary To access a value within the dictionary, use square brackets and the name of the key you wish to access, similar to how we did with lists. For example, for the sample dictionary_1 above entering the key "favorite_food", like so:

```
dictionary_1["favorite_food"]
```

Would return the value "cheeseburger".

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

[11]:
```
#dictionaries
```

[ ]:
```
# my_pets = {'mammals':[my_cat, my_dog], 'birds': 'Chirpy', 'reptiles': {'name':
  ↪ 'Vanessa', 'type': 'gecko', 'age':'1'}}
```

[ ]:
```

```

1.3 Part 2.5 User input

Sometimes in a program we may need to ask for user input, fortunately this is super easy in Python. We simply call the **input()** function:

[12]:
```
#input()

# number_of_pies = input()

# print("the user wants: " + number_of_pies + " pies!")
```

1.4 Part 3: Writing Scripts & writing script output to text files

A small detour into the command line

1.4.1 Scripts

What if we want to write a program or piece of code to be executed later? We can write it in a plain text editor or an IDE (short for 'integrated development environment')! Or write it in chunks here and then transfer it all into a plaintext file. One caveat though, you will need to have at least some basic familiarity with a command line interface (CLI), so an IDE is preferable if you would rather not mess with a CLI.

Jupyter notebooks are great for writing and testing code in small chunks, or for presentations, but if we have a lot of code we want to execute, it's a bit silly to have to execute by hand every code cell we've written into a notebook.

- Write your code in a plain text editor or IDE
- Save file as .py extension, e.g. my_script.py

Then run it straight from the IDE, or from the command line by calling python and giving it the path to your file:

```
python  relative-path-to-file/filename.py
```

It's easiest to run it from the directory that contains the script so that you only have to give the filename, so for our example script, my_script.py, that would look like:

```
python my_script.py
```

1.4.2 Send script output to a text file

One thing to keep in mind when running a script, any output it churns out in the process will be lost if it's not saved! And while Python does have functions for reading and writing text files, the easier thing to do is to redirect the output of the script into a text file from the command line at the time you run the script, like so:

```
python my_scripty.py > my_file.txt
```

1.5 Part 4: Importing packages and modules

Now that you understand the basics of working with data in Python, let's look at importing packages and modules that extend Python's functionality:

1.5.1 Importing Libraries

Often, the tasks you wish to accomplish will require additional functionality beyond what's easily done with base Python. Chances are this functionality exists already in one of the many pre-built libraries available for Python and which we can import into our environment using the **import** command. (Note: if working with Python locally, the libraries must be installed on your system in order to import and use them.)

Some important libraries that you might need depending on application are: - **numpy** : NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more - **pandas** : library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language - **scipy** : SciPy extends numpy and provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems - **beautifulsoup** : a Python library for pulling data out of HTML and XML files - **matplotlib** : Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python - **seaborn** : Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics

There are many other application-specific libraries available, so before you go through the trouble of building up complex functionality from scratch, try Googling or searching in the Python Package Index to see if it already exists: https://pypi.org/

One thing to note is that every one of these libraries will have its own documentation that you will need to read through to learn how to use it.

For the remainder of this tutorial, we will import three libraries: matplotlib, numpy, and pandas. These are the most common libraries for data analysis and visualization.

It is important to always import libraries at the beginning of your code. If the libraries are not imported and you attempt to use them, you will get an error.

Aliases For concision, we will use aliases for each of the libraries: plt for matplotlib, pd for pandas, and np for numpy. Using aliases saves time while coding and makes the code neater, as there are fewer characters.

Note: Make sure the names of the libraries are spelled correctly. If you have any typos while importing a library, Python will return the error, "No module named (insert module)."

Note: if you want to import these into an instance of Python running locally on your machine, you'll need to make sure the package is installed locally in order for Python to be able to access it. We recommend conda as a package and virtual environment manager:

https://docs.conda.io/en/latest/

If you need help with conda, please see our conda tutorial: (Scheduled to be released later this year!)

Guided Exercise: Importing matplotlib, numpy, and pandas

[13]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

1.5.2 Importing only a specific item from a library

If you know that the you'll only need one or two pieces from a library, it's possible to import only those pieces. The advantage to this is not having to call the library name each time. However, this requires that you know the name of the methods you'll be importing from the library.

To do this, the syntax is slightly different than before:

```
from library1 import method1, method2, method3
```

You may have noticed another way to do this when we imported our libraries in the previous step. There, we imported only the *pyplot* module of the matplotlib library by adding the module name connected to the library name with a dot.

As an example we import from the math library the square root and sine functions:

[14]:
```
from math import sqrt, sin, pi
```

[ ]:

[ ]:

1.5.3 Exercise: Downloading and reading a .csv file with the Pandas library

Typically, we have an existing source of data. With the pandas library, we can create a dataframe by downloading and reading a .csv file.

Next in the tutorial, we will be conducting an exploratory analysis of a dataset of Pokemon. This dataset contains information on the first generation of Pokemon, with information on their type and stats.

To read the csv file and store it with a data frame, create a dataframe object (df1) and use **pd.read_csv()**. In the parentheses, add the file path of the .csv file you wish to use. See the Pandas documentation for more info: https://pandas.pydata.org/

df1 = pd.read_csv("file path")

To ensure that the data has loaded correctly, use **df1.head()**, which will show the first 5 rows of the dataset.

Note: Pandas can be very "particular"; it prefers simple .csv files encoded in UTF-8. Depending on the file format and organization of your data, you may need to add additional arguments to get pandas to load and store your data. Double-check that your file path and file name are accurate. For Windows, switch the direction of the slashes within the file path to ensure that it loads correctly.

[15]:
```
#df_pokemon = pd.read_csv("Pokemon.csv")

#df_pokemon.head()
#df_pokemon.tail()
```

There are a number of useful statistical functions from the numpy library that can help with gathering information from your dataset, including:

- mean()
- average() : weighted average
- median()
- std()
- var()

In the next exercise, we will practice using these functions on the Pokemon dataset. To use them on a sequence of numbers, such as a dataframe column, we attach the function to the end of our sequence with a dot '.', for example:

`sequence.var()`

1.5.4 Guided Exercises: Descriptive Statistics with numpy

In the box below, we will practice using each of the statistical functions listed above in conjuction with columns from the dataframe we just created, df_pokemon.

[16]:
```
#take the median of the 'Attack' column
```

[18]:
```
# take the mean of the 'Speed' column
```

[19]:
```
# get the sum of the 'Legendary' column
```

2 That's it for Introduction to Python! :)

2.1 Additional Resources

Congrats, you successfully completed the Introduction to Python training!

Below are additional resources related to Python.

Installing Python

Anaconda: https://www.anaconda.com/download

miniconda: https://docs.anaconda.com/miniconda/miniconda-install/

miniforge: https://conda-forge.org/download/

Python Documentation

https://docs.python.org/3/

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

https://stackoverflow.com/questions/tagged/python

Plaintext Editors with useful features for coding

BBedit (for Mac): https://www.barebones.com/products/bbedit/

Notepad++ (for Windows): https://notepad-plus-plus.org/

Integrated Development Environments (IDEs)

Visual Studio Code: https://code.visualstudio.com/

PyCharm Community Edition: https://www.jetbrains.com/products/compare/?product=pycharm&product=pyc
ce

Matplotlib Documentation and User Guide

https://matplotlib.org/stable/index.html

Pandas User Guide

https://pandas.pydata.org/

NumPy User Guide

https://numpy.org/doc/stable/user/

Python Package Index (PyPI)

https://pypi.org/

PEP 8 Style Guide for Python

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

Python Glossary

https://docs.python.org/3/glossary.html

Github: hosts code projects using Python as well as other languages

https://github.com

[ ]: