Python Basics

Sumner Evans

September 15, 2020

Mines ACM

A Small Survey

Welcome everyone! I'd like to get to know everyone a bit more and get a feel for everyone's prior experience with programming and Python.

- What year are you in?
- How many of you have programmed in any language before?
- How many of you have programmed in **Python** before?

A Small Survey

Welcome everyone! I'd like to get to know everyone a bit more and get a feel for everyone's prior experience with programming and Python.

- What year are you in?
- How many of you have programmed in any language before?
- How many of you have programmed in **Python** before?

A Small Survey

Welcome everyone! I'd like to get to know everyone a bit more and get a feel for everyone's prior experience with programming and Python.

- What year are you in?
- How many of you have programmed in any language before?
- How many of you have programmed in **Python** before?

Overview

What is Python?

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

A Bit of History

- Python first appeared in early 1991. *This means that Python is older than Java and Ruby.*
- Guido van Rossum (GvR, the creator of Python) designed his language with **emphasis on readability**.
- Python was named after *Monty Python's Flying Circus*.
- The language quickly gained popularity because of its appeal to long-time UNIX/C hackers[1].

Why Learn Python?

- Python is a *general purpose, multi-paradigm* language meaning that it is very flexible and can be used in many different scenarios.
- Some of the main applications of Python in industry are web programming, data science, machine learning, automation scripting.
- Python is an easy language to learn.
- Python runs anywhere, and generally requires little setup compared to other languages.

Why Learn Python?

- Python is a *general purpose, multi-paradigm* language meaning that it is very flexible and can be used in many different scenarios.
- Some of the main applications of Python in industry are web programming, data science, machine learning, automation scripting.
- Python is an easy language to learn.
- Python runs anywhere, and generally requires little setup compared to other languages.

Why Learn Python?

- Python is a *general purpose, multi-paradigm* language meaning that it is very flexible and can be used in many different scenarios.
- Some of the main applications of Python in industry are web programming, data science, machine learning, automation scripting.
- Python is an easy language to learn.
- Python runs anywhere, and generally requires little setup compared to other languages.

Why Learn Python?

- Python is a *general purpose, multi-paradigm* language meaning that it is very flexible and can be used in many different scenarios.
- Some of the main applications of Python in industry are web programming, data science, machine learning, automation scripting.
- Python is an easy language to learn.
- Python runs anywhere, and generally requires little setup compared to other languages.

A Note on Python 2 and Python 3

There are two main versions of Python: Python 2 and Python 3. As of earlier this year, Python 2 is no longer supported, so nobody should use it. Unfortunately, many projects and operating systems have not gotten with the times and are still reliant on Python 2.

Python 3 has many major advantages over Python 2 as it fixes many annoying inconsistencies with the older version.

For the purposes of this presentation, we will be talking *strictly of Python 3*.

A Note on Python 2 and Python 3

There are two main versions of Python: Python 2 and Python 3. As of earlier this year, Python 2 is no longer supported, so nobody should use it. Unfortunately, many projects and operating systems have not gotten with the times and are still reliant on Python 2.

Python 3 has many major advantages over Python 2 as it fixes many annoying inconsistencies with the older version.

For the purposes of this presentation, we will be talking *strictly of Python 3*.

A Note on Python 2 and Python 3

There are two main versions of Python: Python 2 and Python 3. As of earlier this year, Python 2 is no longer supported, so nobody should use it. Unfortunately, many projects and operating systems have not gotten with the times and are still reliant on Python 2.

Python 3 has many major advantages over Python 2 as it fixes many annoying inconsistencies with the older version.

For the purposes of this presentation, we will be talking *strictly of Python 3*.

Programming Basics in Python

Follow Along

You can either install Python on your machine or use an online Python environment such as **`https://repl.it/languages/Python3`**.

Most of the things we will cover today can be done directly in the REPL (read-evaluate-print-loop) on the right, however you may want to write code in the file on the left and run it.

Storing Data

At its core, programming is about storing and manipulating *data*.

In almost every programming language, there is a concept of a **variable** which *stores* data.

In Python, you can create a variable using the following syntax:

```
name = "Sumner"
age = 22
likes_acm = True
```

If you have ever programmed in a language such as Java, you may notice that there is no special keyword for declaring a variable.

Storing Data

At its core, programming is about storing and manipulating *data*.

In almost every programming language, there is a concept of a **variable** which *stores* data.

In Python, you can create a variable using the following syntax:

```
name = "Sumner"
age = 22
likes_acm = True
```

If you have ever programmed in a language such as Java, you may notice that there is no special keyword for declaring a variable.

Storing Data

At its core, programming is about storing and manipulating *data*.

In almost every programming language, there is a concept of a **variable** which *stores* data.

In Python, you can create a variable using the following syntax:

```python
name = "Sumner"
age = 22
likes_acm = True
```

If you have ever programmed in a language such as Java, you may notice that there is no special keyword for declaring a variable.

Showing the Data

Storing data isn't any good unless you can actually use it for something useful. One of the most basic things we can do with the data stored is print it out to the console.

To print anything in Python, use the `print` function:

```
name = "Sumner"
age = 22
print(name)
print(age)
```

If you want to print multiple things at once, you can separate them with a comma:

```
print(name, age)
```

Showing the Data

Storing data isn't any good unless you can actually use it for something useful. One of the most basic things we can do with the data stored is print it out to the console.

To print anything in Python, use the `print` function:

```
name = "Sumner"
age = 22
print(name)
print(age)
```

If you want to print multiple things at once, you can separate them with a comma:

```
print(name, age)
```

Showing the Data

Storing data isn't any good unless you can actually use it for something useful. One of the most basic things we can do with the data stored is print it out to the console.

To print anything in Python, use the `print` function:

```python
name = "Sumner"
age = 22
print(name)
print(age)
```

If you want to print multiple things at once, you can separate them with a comma:

```python
print(name, age)
```

Getting Data From the User

Often, we want to get some input from the user and store it in a variable. To do this in Python, we use the `input` function.

```
name = input()
print("Hello", name)
```

We can also optionally include prompt text:

```
age = int(input("How old are you? "))
print("In one year, you will be", age + 1, "years old")
```

What do you think the difference between `input()` and `int(input())` is?

Getting Data From the User

Often, we want to get some input from the user and store it in a variable. To do this in Python, we use the `input` function.

```
name = input()
print("Hello", name)
```

We can also optionally include prompt text:

```
age = int(input("How old are you? "))
print("In one year, you will be", age + 1, "years old")
```

What do you think the difference between `input()` and `int(input())` is?

Getting Data From the User

Often, we want to get some input from the user and store it in a variable. To do this in Python, we use the `input` function.

```
name = input()
print("Hello", name)
```

We can also optionally include prompt text:

```
age = int(input("How old are you? "))
print("In one year, you will be", age + 1, "years old")
```

What do you think the difference between `input()` and `int(input())` is?

Getting Data From the User

Often, we want to get some input from the user and store it in a variable. To do this in Python, we use the `input` function.

```
name = input()
print("Hello", name)
```

We can also optionally include prompt text:

```
age = int(input("How old are you? "))
print("In one year, you will be", age + 1, "years old")
```

What do you think the difference between `input()` and `int(input())` is?

What Sorts of Data Can We Store?

There are many different *types* of data that we can store in Python. Here are the most basic data types (primitives):

- `bool` — either `True` or `False`
- `int` — an integer
- `float` — a real number[2]
- `string` — a sequence of characters[3]

[2]Not all real numbers can be represented as a floating point number, but that's not normally important.
[3]Note that unlike other languages, there is no `char` datatype. Chars are just one-character strings.

Manipulating Data: Assigning a New Value to a Variable

Having variables to store is nice, but a lot of times we want to modify the value stored in the variable!

To do that, we use a very similar syntax to defining variables:

```
name = "Sumner"  # declares a variable and gives it an
                 # initial value
print(name)

name = "Jack"    # assigns the value "Jack" to the "name"
                 # variable
print(name)
```

Manipulating Data: Basic Operations

Similar to how you can perform operations on variables in algebra, you can perform operations on variables. Here are some basic operations on primitive data types in Python:

- `+`, `-`, `*`, `/`, `//`: add, subtract, multiply, divide, integer division.
- `**`: exponentiate (3^8 would be written `3**8`).
- `<`, `>`, `==`: tests if two numbers are less than, greater than, equal to each other, respectively.

Try it Yourself!

Try creating a few different variables of various types and performing operations on them.

A few examples to get you started:

```
pi = 3.14159265
r = 10
print("diameter =", pi * (r ** 2))

email = input("Enter your email: ")
email_again = input("Verify your email: ")
print(email == email_again)
```

Making Decisions: Selection Using `if`

It’s all well and good that we can compare numbers and get a boolean value, but we need to make *decisions* with that information. That’s where **`if` statements** come in.

The syntax for `if` statements in Python is:

```
if condition1:
    # do whatever is in this indented section if condition1
    # is True
elif condition2:
    # do whatever is in this indented section if condition2
    # is True
else:
    # do whatever is in this indented section otherwise
```

You can have arbitrary many `elif` blocks (including no elif blocks). It is also not necessary to have an `else` block.

Making Decisions: Selection Using `if`

It's all well and good that we can compare numbers and get a boolean value, but we need to make *decisions* with that information. That's where **`if` statements** come in.

The syntax for `if` statements in Python is:

```
if condition1:
    # do whatever is in this indented section if condition1
    # is True
elif condition2:
    # do whatever is in this indented section if condition2
    # is True
else:
    # do whatever is in this indented section otherwise
```

You can have arbitrary many `elif` blocks (including no elif blocks). It is also not necessary to have an `else` block.

Making Decisions: Selection Using `if`: Example

Here is a simple example of an `if` statement at work:

```python
likes_python = input("Do you like Python? (y/n)")
if likes_python == "y":
    print("You like Python!")
elif likes_python == "n":
    print("You don't like Python :(")
else:
    print("I don't know if you like Python...")
```

Making Decisions: Selection Using `if`: Your Turn!

Now it's your turn! Try to write some Python which does the following:

1. Creates a variable with a secret number of your choice.
2. Asks the user to guess a number.
3. Tells the user if their guess is above, below, or equal to the secret number.

Extra Credit: Look up the documentation for the `random.randint` function and see if you can make the secret number random.

Doing Things Many Times: `for` loops

Often, we might want to do the same thing (or a similar thing) multiple times. In these situations, we need *loops*.

The most common type of loop in Python is the `for` loop. Unlike other languages, Python's `for` loop is a *range-based for loop*:

```
for x in <iterable>:
    # do what is in this indented block
```

Whatever is in the *indented* section will be run for every value in the iterable.

Doing Things Many Times: `for` loops

Often, we might want to do the same thing (or a similar thing) multiple times. In these situations, we need *loops*.

The most common type of loop in Python is the **`for`** loop. Unlike other languages, Python's **`for`** loop is a *range-based for loop*:

```
for x in <iterable>:
    # do what is in this indented block
```

Whatever is in the *indented* section will be run for every value in the iterable.

Doing Things Many Times: `for` loops: Example

There are a lot of things in Python are iterable. We will see more of them soon, but here's an easy example to get you started:

```python
for x in range(20):
    print(x)
```

Any ideas what this will print?

The syntax for range is `range(start, stop, step)`

- `start` is the number to start on
- `stop` is the number to stop *before*
- `step` is the amount to increment each time

Both `start` and `step` are optional and if omitted, they will be assumed to be `0` and `1`, respectively.

Doing Things Many Times: `for` loops: Example

There are a lot of things in Python are iterable. We will see more of them soon, but here's an easy example to get you started:

```python
for x in range(20):
    print(x)
```

Any ideas what this will print?

The syntax for range is `range(start, stop, step)`

- `start` is the number to start on
- `stop` is the number to stop *before*
- `step` is the amount to increment each time

Both `start` and `step` are optional and if omitted, they will be assumed to be `0` and `1`, respectively.

Doing Things Many Times: `while` loops

Sometimes we don’t know how many times we need to run a block of code. In these cases, we can use a `while` loop.

The syntax for a `while` loop is:

```
while <condition>:
    # do what is in this indented block
```

Doing Things Many Times: `while` loops

Sometimes we don't know how many times we need to run a block of code. In these cases, we can use a `while` loop.

The syntax for a `while` loop is:

```
while <condition>:
    # do what is in this indented block
```

Doing Things Many Times: `while` loops: Example

What do you think this snippet of code will do?

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Doing Things Many Times With Loops: Your Turn!

Now it's your turn! Try to write some Python which does the following (this should feel similar to what you did with the `if` statement):

1. Creates a variable with a secret number of your choice.
2. Asks the user to guess a number.
3. Tells the user if their guess is above, below, or equal to the secret number.
4. **Keeps asking the user to guess until they get the right number.**

Extra Credit: Look up the documentation for the `random.randint` function and see if you can make the secret number random.

Containers for Data: Abstract Data Types

What is an Abstract Data Type?

Often, we need to store multiple pieces of data together in a *data structure*. Some examples of when this would be useful are:

- Storing a list of all of the students in a class.
- Storing the grades associated with each student in a class.

The reason they are called abstract data types is that you don't need to know how they are implemented, you just need to know how they work.

What is an Abstract Data Type?

In Python, we have four main data structures:

- **Lists** — store an *ordered* set of elements that *can* be changed.
- **Tuples** — store an *ordered* set of elements that *cannot* be changed.
- **Sets** — store an *unordered*, *deduplicated* set of elements that *can* be changed.
- **Dictionaries** — store an unordered set of mapping/associations of *keys* to *values*.

Lists

Lists store an ordered set of elements that can be changed. You can declare a list *literal* like this:

```
[1, 2, 3]
```

If you want to *access* a specific element of the list, you can *index* into the list like so:

```
my_list = [1, 2, 3]
print(my_list[0])
my_list[0] = 4
print(my_list)
```

Note: indexes start at `0` (they are zero-indexed).

If you want to add to a list, you can use the `append` function:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

Lists

Lists store an ordered set of elements that can be changed. You can declare a list *literal* like this:

```
[1, 2, 3]
```

If you want to *access* a specific element of the list, you can *index* into the list like so:

```
my_list = [1, 2, 3]
print(my_list[0])
my_list[0] = 4
print(my_list)
```

Note: indexes start at `0` (they are zero-indexed).

If you want to add to a list, you can use the `append` function:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

Lists

Lists store an ordered set of elements that can be changed. You can declare a list *literal* like this:

```
[1, 2, 3]
```

If you want to *access* a specific element of the list, you can *index* into the list like so:

```
my_list = [1, 2, 3]
print(my_list[0])
my_list[0] = 4
print(my_list)
```

Note: indexes start at `0` (they are zero-indexed).

If you want to add to a list, you can use the `append` function:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

Tuples

Tuples store an ordered set of elements that cannot be changed. You can declare a tuple *literal* like this:

```
(1, 2, 3, 4)
```

Like lists, you can access individual elements of the tuple in the same way as a list.

```
my_tuple = (1, 2, 3)
print(my_tuple[0])
```

You cannot modify a tuple, but you can add them together (you can actually do this with lists, too!):

```
my_tuple1 = (1, 2, 3)
my_tuple2 = (4, 5, 6)
print(my_tuple1 + my_tuple2)
```

Tuples

Tuples store an ordered set of elements that cannot be changed. You can declare a tuple *literal* like this:

```
(1, 2, 3, 4)
```

Like lists, you can access individual elements of the tuple in the same way as a list.

```
my_tuple = (1, 2, 3)
print(my_tuple[0])
```

You cannot modify a tuple, but you can add them together (you can actually do this with lists, too!):

```
my_tuple1 = (1, 2, 3)
my_tuple2 = (4, 5, 6)
print(my_tuple1 + my_tuple2)
```

Tuples

Tuples store an ordered set of elements that cannot be changed. You can declare a tuple *literal* like this:

```
(1, 2, 3, 4)
```

Like lists, you can access individual elements of the tuple in the same way as a list.

```
my_tuple = (1, 2, 3)
print(my_tuple[0])
```

You cannot modify a tuple, but you can add them together (you can actually do this with lists, too!):

```
my_tuple1 = (1, 2, 3)
my_tuple2 = (4, 5, 6)
print(my_tuple1 + my_tuple2)
```

Sets

Sets store an *unordered, deduplicated* set of elements that can be changed. You can declare a set *literal* like this:

```
{1, 2, 3, 2}
```

Note this is equivalent to `{1, 2, 3}` because of the deduplication.

There are two main operations on sets: `add` and `remove`:

```
my_set = {1, 2, 3}
my_set.add(4)
my_set.remove(1)
print(my_set)
```

Sets

Sets store an *unordered, deduplicated* set of elements that can be changed. You can declare a set *literal* like this:

```
{1, 2, 3, 2}
```

Note this is equivalent to `{1, 2, 3}` because of the deduplication.

There are two main operations on sets: `add` and `remove`:

```
my_set = {1, 2, 3}
my_set.add(4)
my_set.remove(1)
print(my_set)
```

Lists, Tuples, and Sets + `for` Loops

Lists, Tuples, and Sets are all iterable, meaning they can be used in a `for` loop:

```
my_list = [1, 2, 3]
for x in my_list:
    print(x)
```

Your Turn: Try and create a list of numbers, then for each element of the list, print whether that number is less than, equal to, or greater than `42`.

Now try the same thing with a tuple and a set.

Lists, Tuples, and Sets + `for` Loops

Lists, Tuples, and Sets are all iterable, meaning they can be used in a `for` loop:

```python
my_list = [1, 2, 3]
for x in my_list:
    print(x)
```

Your Turn: Try and create a list of numbers, then for each element of the list, print whether that number is less than, equal to, or greater than `42`.

Now try the same thing with a tuple and a set.

Lists, Tuples, and Sets + `for` Loops

Lists, Tuples, and Sets are all iterable, meaning they can be used in a `for` loop:

```
my_list = [1, 2, 3]
for x in my_list:
    print(x)
```

Your Turn: Try and create a list of numbers, then for each element of the list, print whether that number is less than, equal to, or greater than `42`.

Now try the same thing with a tuple and a set.

Dictionaries

Dictionaries store an unordered set of mapping/associations of *keys* to *values*. You can declare a set *literal* like this:

```
{"key": "value", 3: 2, 3: 8}
```

Note this is equivalent to `{"key": "value", 3: 8}` because, like sets, the *keys* are deduplicated.

You can access the value associated with a key of a dictionary similar to how you access an element of a list:

```
my_dict = {"key": "value", 3: 8}
print(my_dict["key"])
```

You can also use similar syntax to edit entries in the dictionary and add new entries.

```
my_dict = {"key": "value", 3: 8}
my_dict["new key"] = 42
my_dict["key"] = 18
print(my_dict)
```

Dictionaries

Dictionaries store an unordered set of mapping/associations of *keys* to *values*. You can declare a set *literal* like this:

```
{"key": "value", 3: 2, 3: 8}
```

Note this is equivalent to `{"key": "value", 3: 8}` because, like sets, the *keys* are deduplicated.

You can access the value associated with a key of a dictionary similar to how you access an element of a list:

```
my_dict = {"key": "value", 3: 8}
print(my_dict["key"])
```

You can also use similar syntax to edit entries in the dictionary and add new entries.

```
my_dict = {"key": "value", 3: 8}
my_dict["new key"] = 42
my_dict["key"] = 18
print(my_dict)
```

Dictionaries + `for` Loops

Dictionaries are also iterable but most of the time what you care about iterating over are either all of the keys, values, or pairs:

```python
my_dict = {"key": "value", 3: 8}
print(my_dict.keys())
print(my_dict.values())
print(my_dict.items())


for key, value in my_dict.items():
    print(key, value)
```

Your Turn: Create a dictionary containing only numeric keys and values and write a `for` loop which adds the key to the value for each of the pairs.

Dictionaries + `for` Loops

Dictionaries are also iterable but most of the time what you care about iterating over are either all of the keys, values, or pairs:

```
my_dict = {"key": "value", 3: 8}
print(my_dict.keys())
print(my_dict.values())
print(my_dict.items())


for key, value in my_dict.items():
    print(key, value)
```

Your Turn: Create a dictionary containing only numeric keys and values and write a `for` loop which adds the key to the value for each of the pairs.

Functions

What is a Function?

A lot of times, we have code that we want to run multiple times, but not necessarily in right in a row on the same data structure. In these cases, we can use *functions* to organize our code.

You've actually already seen functions! `print`, `input`, `append`, and `items` are all functions!

But we can write our own functions as well.

What is a Function?

A lot of times, we have code that we want to run multiple times, but not necessarily in right in a row on the same data structure. In these cases, we can use *functions* to organize our code.

You've actually already seen functions! **`print`**, **`input`**, **`append`**, and **`items`** are all functions!

But we can write our own functions as well.

Defining our own Functions

The syntax for defining a function is:

```
def my_function(parameter1, parameter2, ...):
    # do things involving the parameters
    return something
```

A few notes:

- You can have as many parameters as you want (including none).
- You do not have to return anything. (By default, the function will return `None`.)

Defining our own Functions: Example

Here's a simple function which adds two numbers together:

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

Functions: Your Turn!

- Create a function that takes in a list as a parameter and returns the list, backwards.
- Create a function that takes two tuples representing two `(x, y)` coordinates and return the *Cartesian distance* between the two points. The formula is:

  $$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  (Look up the `math.sqrt` function.)

Classes

Classes: The Basics

There are a lot of things that you can do with classes, but for Carrier Python, most of what we use them for is to organize variables and functions into logical containers. Here’s an example class:

```
class MyClass:
    counter = 0

    def add_to_variable(self, amount):
        self.variable = self.variable + amount
```

You can then instantiate a class like so:

```
my_object = MyClass()
print(my_object.counter)
my_object.add_to_variable(4)
print(my_object.counter)
```