Welcome to Python!

Hello reader,

This document will jumpstart your understanding of Python. We will give a broad overview to all the foundational aspects of the programming language. Feel free to come back to this section - after all, even the most experienced programmers have to review the basics sometimes!

Data Types

Programming languages give us an easy way to manipulate data in the computer - but first, the computer has to understand what kind of data we're dealing with!

Here are some examples of the kinds of data you can work with in the Python programming language:

Table 1: Data Types in Python

Data Type	Examples
Integers	`-3, -2, -1, 0, 1, 2, 3`
Floating-point numbers	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`
Strings	`'a', 'Hello World!', "C-3PO", "bbb"`
Boolean	`True, False`

Integers and floating-point numbers are also called *numeric types*. We shall cover these more in depth now.

Numeric Types

Numbers are everywhere! In Python, we have 3 kinds of data types to represent numerical data:

- `int` - whole numbers
- `float` - floating-point (aka fractional) numbers

Between the `int` and `float` types, Python 3 lets you store any kind of real number in a variable.

Note: there is also a `complex` data type in Python - you may read about it in the documentation.

The numeric types are useful for when you are doing mathematical operations - which we will discuss next!

Math Operations Python supports a lot of these! Table 2 describes which operator you can use for each of them, along with examples. ##### Table 2: Mathematical Operators (for Each Operation)

Operator	Operation	Example
**	Exponent	`3 ** 3 = 27`
%	Modulus/Remainder Division - *returns only the remainder*	`30 % 8 = 6`
//	Integer division - *truncates any floating-point values in the quotient*	`30 // 8 = 3`
/	Division - *always returns a* `float`	`30 / 8 = 3.75`
*	Multiplication	`2 * 3 = 6`
-	Subtraction	`7 - 2 = 5`
+	Addition	`9 + 2 = 11`

Note: all the operators above will return: - a `float`, if the two operands are 1) both floating-point OR 2) they have different types - an `int`, if 1) both operands are integers AND 2) it is not the division (`/`) operator

Order of Operations Python prioritzes some operations over others. This means that it follows a *precedence order* that decides which operation to compute first, when there are several of them together in a given expression.

The operators in Table 2 are listed from **highest** to **lowest** precedence, for your reference.

The next data type we will cover are strings, which are also known as a *text sequence type*.

Strings

Strings in Python are used to represent any sequence of characters. Basically, anything that can be typed on a computer between a pair of quotes is a string!

Common String Operations

1. **String concatenation**:

   Concatenation is when one string is added onto the end of another string. A quick and easy way to do this is by using the addition (`+`) operator, just like we did for numbers!

For example, let's say we wanted a string that said `'Alice met Allison on Tuesday'` - then the code could look like the following:

```
'Alice met ' + 'Allison on Tuesday'
```

Note: although using addition works, it is far from perfect. In practice we prefer 1) *string formatting*, which will be discussed later in this book, or 2) the `str.join()` function, which you may read about in the Python documentation.

2. **String Replication**:

   Replication is when we repeat a given string over and over again in a new, longer `str` object.

   Incidently, Python makes this easier for us through the `*` operator.

   ```
   'Alice' * 5
   ```

3. **Displaying Messages using `print()`** You will most often see strings where a program needs to tell the user some kind of message - this is exactly what the `print()` function will do:

   ```
   print('Hello world!')
   ```

4. **Gathering User Responses using `input()`**

   Another common task is for a computer program to ask the user for their information - this can be done in Python via the `input()` function.

   Consider the following example:

   ```
   name = input('What is your name?') # ask for their name
   ```

   The user will first be prompted to enter their name, and then their response is captured in the variable called `name`.

5. **Measuring Length using `len()`**

   We can also calculate the number of characters in a string:

   ```
   len('hello')  # result: 5
   ```

   The `len()` function is also applicable to all of Python's "container" data types, which are used to store multiple pieces of data at the same time (like lists, sets, and dictionaries)!

Common Mistakes with Strings:

Strings have such a wide definition, that it is easy to make mistakes when working with them. Here are some things to watch out for:

1. Not enclosing them in single quotes (e.g. `'Hello'`) or double quotes (e.g. `"World"`)
2. Although Python treats single and double quotes the same, you *cannot* combine the two (e.g. do NOT do `"Hello, World!'`)

Quotes in Strings You are still allowed to mix double quotes and single quotes. As long as you enclose each pair of quotes, they can in fact be very useful to use together - such as when a string includes a quote!

Here's what NOT to do:

```
print("I said "hello."")  # ERROR!
```

In the above code snippet, the `'hello'` ends up not being enclosed by any pair of quotes.

So how to remedy this? One option is to enclose the whole string in a pair of *single quotes*, so we can let the *double quotes* in the sentence enclose the `'hello'` properly:

```
print('I said "hello."')  # all good :)
```

Next, let's take a look at the Boolean data type!

Boolean Types

The Boolean data type, represented as `bool` in Python, needs no overthinking - it simply represents whether `True` or `False`.

As you might imagine, they are especially useful for writing code that answers yes/no questions - we will discuss this more in-depth when we talk about Control Flow.

The `str()`, `int()`, and `float()` Functions

Python's data types are not necessarily set in stone. In some scenarios (shown below) we can convert, or *cast*, a given value in one data type to another, by calling the desired data type like a function! More technically, this is also called *explicit type conversion*.

Variables

The data in our program can change not only in its **type**. In addition, it is even more common for our data to change in its **value**. A *variable* in programming is how we represent some piece of data whose value can be many different things.

You can name a variable anything as long as it obeys the following rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (`_`) character.
3. It can't begin with a number.
4. Variable name starting with an underscore (`_`) are considered as "unuseful'.

Augmented Assignment Operators

Python provides additional operators that provide us with a shorthand for when we are updating the value inside a variable. Table 8 describes these:

Table 8: Augmented Assignment Operators (and the Equivalent Statements)

Operator	Equivalent
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

Control Flow

While knowing how to store data is nice, it is even cooler to be able to write programs that can let the computer do certain tasks for us.

Control flow refers to how the computer figures out what to do with our data in a given situation. This is most often done through giving the computer specific conditions on what to do in a given scenario ahead of time.

For example, you might decide whether to wear sunglasses or not, based on if the sun is shining. If we were to write *pseudocode* for this decision, it might look like the following:

```
if it's sunny outside,
    THEN, I will wear_sunglasses()!
```

Let's see how to model this in Python code - the first piece to making it possible is *comparison operators*!

Comparison Operators

Comparison operators help our program decide whether or not a given piece data meets a given condition.

Table 3 shows the comparison operators found in Python: ##### Table 3: Comparison Operators and What They Mean

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater Than

Operator	Meaning
`<=`	Less than or Equal to
`>=`	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them. They all follow a syntax of `data <operator_goes_here> condition` (but you don't necessarily have to put the `data` on the left side and `condition` on the right).

Note: since the result of a comparison operation is only True or False, this means comparison operators *must* return Boolean(`bool`) values.

Boolean Operators

There are also operators specifically meant for working with `bool` values. Specifically, the four Boolean operators are `is`, `and`, `or`, and `not`.

Tables 4-7 show where each of these operators returns `True` or `False`:

Table 4: The *is* Operator's *Truth* Table:

Expression	Evaluates to
True is True	True
True is False	False
False is True	False
False is True	True

Table 5: The *and* Operator's *Truth* Table:

Expression	Evaluates to
True and True	True
True and False	False
False and True	False
False and False	False

Table 6: The *or* Operator's *Truth* Table:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True

Expression	Evaluates to
False or False	False

Table 7: The *not* Operator's *Truth* Table:

Expression	Evaluates to
not True	False
not False	True

Boolean evaluation

By convention, Python programmers do not like to use the `==` or `!=` operators to evaluate `bool` values themselves. Instead, it is more popular to use the `is` or `is not` operators.

Mixing Boolean and Comparison Operators

Python is not *totally* against us mixing Boolean and comparison operators. See if you can correctly identify the return value of each expression:

```
(5 >= 7) and (5 < 6)
```

```
(1 == 2) or (2 == 2)
```

Note: although we are mixing Boolean and comparison operators here, notice how we use parentheses `()` to clearly show the how the Boolean operators `or` and `and` are only working with values that have already been evaluated to be `True` or `False`.

if Statements

The `if` statement takes the value computed by a Boolean evaluation, and if it is `True` we can run a specific piece of code to so our program does the appropiate thing in that scenario.

Going back to our sunglasses example above, we are one step closer to writing code that can tell our user when to put on sunglasses:

```
if is_sunny == True:
    print("You're going to need sunglasses today!")
```

else Statements

The `else` statement is a complement to the `if` statement - that is, if the condition in the `if` statement evaluates to `False`, then control flow in our program will go and execute the code found indented below our `else` keyword:

```
if is_sunny == True:
    print("You're going to need sunglasses today!")
else:  # the sun is NOT out
    print("You're all set :)")
```

elif Statements

The `elif` keyword is short for "else if". Like the `else` statement, it comes after an `if` statement and will be evaluated by control flow in our program, if the condition in our `if` statement is `False`. BUT, it can also be given a condition of its own, just like the `if` statement! to allow for writing more nuanced programs!

Implicit Boolean Evaluation

It can be cumbersome to write out the full condition for an `if` or `elif` statement all the time - and that's why Python actually lets us take it out if we want!

All of the `if` statements in the code below are equivalent:

```
a = (2 + 2 == 4)

if a is True:
   pass
if a is not False:
   pass
if a:
   pass
```

while Loop Statements

The `while` loop is used just the `if` statement, except that it will exhaustively execute the code in its block. It will only stop once the condition it has evaluates to `False`.

for Loops and the range() Function

Another kind of loop in Python is the `for` loop. Unlike the `while` loop, it doesn't need a condition - instead, `for` loops are used when we are certain about how many *iterations*, or the number of times our code executes, that we want to have.

Therefore, the syntax of the `for` loop is only there to let the program know how many iterations to go through.

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times ({})'.format(str(i)))
```

If you run the code snippet above, you will see the following output:

```
'Jimmy Five Times (0)'
'Jimmy Five Times (1)'
'Jimmy Five Times (2)'
'Jimmy Five Times (3)'
'Jimmy Five Times (4)'
```

The `range()` function is commonly used in `for` loops, and it is a good one to remember - it essentially is used to set the value of `i` on each of the 5 iterations our `for` loop goes through. You may also read more about it on the Python documentation as you wish.

The `range()` function can also be called with three arguments. The first two arguments will be the *start* and *stop* values, and the third will be the *step* argument. The step is the amount that the variable is increased by after each iteration.

Once the value of our *iterator* (in this case `i`) variable reaches or surpasses the value of the *stop* argument, our for loop terminates:

Can you tell how many iterations this loop has?

```
for i in range(0, 10, 2):
   print(i)
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

Container Types

We saw above that by using variables, we can save certain pieces of data in our Python program, to be used later. But, what if we suddenly had many different pieces of information to keep track of?

Lists in Python

Lists (represented by the `list` type) simply represent an *ordered sequence of values*.

Just as strings are a sequence of textual characters, the `list` in Python allows us to collect lots of different values altogether sequentially, enclosed in a pair of square brackets (`[]`).

Getting Individual Values in a List with Indexes Because the data in a Python `list` is in a sequence, we can retrieve individual values in that `list` (aka an *element of the list*) if we know where it is positioned in that list.

The position of an element in a `list` is the *index* of that element. The index position of the first element is `0`, and the indices of the following elements incrementally increases by 1.

Negative Indexes Python will also let us access elements from the back of the list, using negative index values. These begin with the value of `-1` for the very last element, and decrease by one as we move further to the left.

Getting Sublists with Slices We can retrieve several items out of a `list` at once and put them in a new `list` by using a *slice* with the `:` operator.

Slices have a similar syntax to the `range()` function - you can optionally pass in `start:stop:step` values when you use it, or use the default values.

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam[0:4:2]  # result: ['cat', 'rat']

spam[1:3]  # result: ['bat', 'rat']

spam[0:-1]  # result: ['cat', 'bat', 'rat']

spam[:]  # result: ['cat', 'bat', 'rat', 'elephant']
```

Removing Values from Lists One common way to do this is with the `del` keyword. Once an element from `list` is gone, all the elements that came after it will move over an index postion to the left.

```
spam = ['cat', 'bat', 'rat', 'elephant']
del spam[2]
spam  # result: ['cat', 'bat', 'elephant']
```

Loops and Lists Once we have a list, we can `for` loop if we want to do something on each of the elements. This is known as a *list traversal*.

When we use the syntax `for iterator in my_list`, our `iterator` variable will be set equal to the value of each element in `my_list`, starting from the leftmost index. In the body of the for loop, we can then do what we like with that value:

```
supplies = ['pens', 'staplers', 'flame-throwers', 'binders']

for i, supply in enumerate(supplies):
    print('Index {} in supplies is: {}'.format(str(i), supply))
```

The in and not in Operators Python strives to be as close to readable English as possible. The `in` keyword was added to the language in order to help us deterimine if a given value is present in a container data type in Python. In the context of lists, it will return `True` or `False` depending on if the given value is equal to one of the elements in the given list:

```
'howdy' in ['hello', 'hi', 'howdy', 'heyas']
```

This is also known as checking for *membership* of the data in the list. If we want to, we can also place `not` before the `in` operator, to check if the given value "is not" a member of that list.

Finding a Value in a List with the index Method Once we know a value exists in a list, wouldn't it be great to know what it's index position was?

```
spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
spam.index('Pooka')
```

Adding Values to Lists with append and insert Values can be added to the end of a list with the `append()` method:

append():

```
spam = ['cat', 'dog', 'bat']
spam.append('moose')
spam
```

insert(): Values can be added to a specific index in the list using `insert()`:

```
spam = ['cat', 'dog', 'bat']
spam.insert(1, 'chicken')
spam
```

Removing Values from Lists with remove The `remove()` helps us delete a particular element from a list. It is different from using the `del` keyword, since we do not need to know the index of the element we want to delete:

```
spam = ['cat', 'bat', 'rat', 'elephant']
spam.remove('bat')
spam
```

If the value appears multiple times in the list, only the first instance of the value will be removed.

Sorting the Values in a List with sort The `sort()` functions are provided so we can easily order the elements of a list from least to greatest (if they are numerical values) or in lexicographical order (if they are strings).

Invoking the `.sort()` method on an already existing `list` object will modify the elements of that list itself.

You can also pass True for the `reverse` keyword argument to have sort() sort the values in reverse order.

If you need to sort the values in regular alphabetical order, pass str. lower for the key keyword argument in the sort() method call:

```
spam = ['a', 'z', 'A', 'Z']
spam.sort(key=str.lower)
spam
```

You can use the built-in function `sorted` to return a new list:

```
spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
sorted(spam)
```

Dictionaries and Structuring Data

The next kind of container data type is the *dictionary* (represented as `dict` in Python). A dictionary is used to store an unordered collection of key-value pairs.

The fact it is *unordered* means we cannot obtain values from a `dict`. However, all the keys in a `dict` are unique - so we can get a specific value from a `dict`, if we know the corresponding key.

Example Dictionary:

```
myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

The keys, values, and items Methods values(): This function returns only the values in a dictionary:

```
spam = {'color': 'red', 'age': 42}

for v in spam.values():
    print(v)
```

keys(): This returns just the keys:

```
for k in spam.keys():
    print(k)
```

items(): This will return both the keys, and their corresponding values:

```
spam = {'color': 'red', 'age': 42}

for k, v in spam.items():
    print('Key: {} Value: {}'.format(k, str(v)))
```

Checking if a Key or Value Exists in a Dictionary The `in` operator can still be used with dictionaries:

```
spam = {'name': 'Zoey', 'age': 7}
'name' in spam.keys()  # equal to 'name' in spam
```

You can omit the call to keys() when checking for a key:

```
'color' in spam
```

But not when checking for values:

```
'Zophie' in spam.values()
```

The get Method If you try to access a value in a `dict` with a key that isn't actually in that dictionary, the Python interpreter will throw a `KeyError` exception. To avoid this, we can use the `get()` method.

With this method, we can specify both the key for the value we want from the `dict`, AND also a default value as a second parameter. Therefore, if the key isn't actually in the `dict`, the method will instead return whatever default argument we passed in:

```
picnic_items = {'apples': 5, 'cups': 2}
'I am bringing {} cups.'.format(str(picnic_items.get('cups', 0)))
```

The setdefault Method The `setdefault()` method will add a key-value pair to our dictionary, if it doesn't already already exist.

Let's consider this code. It is possible to add/update key-value pairs in a `dict` using square brackets (`[]`):

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

Using `setdefault` we could make the same code more succinctly:

```
spam = {'name': 'Pooka', 'age': 5}
spam.setdefault('color', 'black')
```

Finallly, the last container data type we will discuss are sets! ### Sets

Sets are an unordered collection of elements. You can think of sets as 1-sided dictionaries - specifically, if dictionaries are collections of key-value pairs, then sets are what you would have if you only had the keys!

Sets are mostly often used for membership testing, and eliminating duplicate entries. Set objects also support mathematical operations like *union*, *intersection*, *difference*, and *symmetric difference*.

Initializing a set There are two ways to create sets: using curly braces `{}` and the bult-in function `set()` around a list:

```
s = {1, 2, 3}
s = set([1, 2, 3])
```

When creating an empty set, be sure to not use the `set()` function (you will get an empty dictionary if you use `{}` instead).

```
s = {}
type(s)
```

sets: unordered collections of unique elements A set automatically remove all the duplicate values.

```
s = {1, 2, 3, 2, 3, 4}
s
```

Adding Elements to a Set Using the `add()` method we can add a single element to the set.

```
s = {1, 2, 3}
s.add(4)
s
```

And with `update()`, multiple ones .

```
s = {1, 2, 3}
s.update([2, 3, 4, 5, 6])
s  # remember, sets automatically remove duplicates
```

Removing Elements From a Set Like lists, set provide a `remove()` to help us delete elements from it.

Alternatively, you can also use the `discard()` method, which won't raise any errors if the argument passed is not actually in the `set`.

```
s = {1, 2, 3}
s.discard(3)
s

s.discard(3)
```

Set Operations:

1. `union()` or `|` will create a new set that contains all the elements from the sets provided.

   ```
   s1 = {1, 2, 3}
   s2 = {3, 4, 5}
   s1.union(s2)  # or 's1 | s2'
   ```

2. `intersection` or `&` will return a set containing only the elements that are common to all of them.

   ```
   s1 = {1, 2, 3}
   s2 = {2, 3, 4}
   s3 = {3, 4, 5}
   s1.intersection(s2, s3)  # or 's1 & s2 & s3'
   ```

3. `difference` or `-` will return only the elements that are in one of the sets.

   ```
   s1 = {1, 2, 3}
   s2 = {2, 3, 4}
   ```

```
s1.difference(s2)  # or 's1 - s2'
```

4. `symetric_difference` or `^` will return all the elements that are *not* common between them.

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
s1.symmetric_difference(s2)  # or 's1 ^ s2'
```

Now that you about the most common data types in Python, you can see how we can start to store sizeable portions of data together for our programs. The last skill you will need before you can actually begin solving data science problems with Python is knowing how to work with *functions* - let's discuss these next!

Functions

Functions are blocks of code which we can use to perform a certain task over and over again.

The first line of a function is its *declaration*. It includes the `def` keyword (to tell Python we are "defining" a function), the name of the function (which follows the naming rules as variables do), and then whatever *parameters* the function has go inside a pair of parentheses.

Parameters are nothing to be afraid of - they are special variables that can be used once the function is actually called! ### Return Values and return Statements

When creating a function using the `def` keyword, you can specify what the return value should be with a `return` keyword. A return statement consists of the following:

- The `return` keyword.
- The value or expression that the function should return.

Once the flow of control hits a `return` statement, it will execute that line of code and exit the function.

```
import random
def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
```

```
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'
```

The None Value

Some functions have no `return` statement - these functions have a return value of `None`, which is just Python's way of representing the *absence* of data.

For example, the `print()` function in Python has a return value of `None`:

```
spam = print('Hello!')
spam is None
```

Note: never compare to `None` with the `==` operator. Always use `is`.

Local and Global Scope

All the variables in a Python program have a *scope*. *Scope* is how we describe the place in a program where we can actually access the value of a variable.

To start off with, let's discuss the **2 main scopes in Python:**

1. *Local*: we say all the variables *inside* of a function have a *local* scope - meaning, they can be only be accessed within the function body.
2. *Global*: these are all the variables outside of any function.

Rules of Thumb For Scope:

Here are some basic rules to remember, so you never run into issues with scope:

1. The code in the global scope cannot use any local variables.
2. However, a variable in a local scope can access global variables.
3. The code in a function's local scope cannot use variables in any other function.
4. You *can* use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

What Scope Does This Variable Have?

There are four rules to tell whether a variable is in a local scope or global scope:

1. If the variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, then it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, then it is a local variable.
4. However, if the variable is not used in an assignment statement, then it is a global variable.

Credits

This section would not be possible if not for help from the following sources:

1. *Automate the Boring Stuff with Python*, a book by Al Sweigart under the Creative Commons license.
2. The Python Cheatsheet website and GitHub repository by wilfredinni. {“mode”:“full”,“isActive”:false}