Python Guide
Chapters 1-2, Basics of Python

Daniel Scarbrough

February 14, 2021

Contents

Chapter 1

Introduction

In Chapters 1 and 2, you will be introduced to the very basics of Python through a few examples. Each file in these examples is included in this repository and can be viewed, downloaded, copied, edited, etc. When first learning it can be helpful even to copy examples from scratch - meaning not using copy-paste - but typing it out line for line. Python is a language, and much like learning a spoken language this type of practice is helpful to start becoming fluent.

The examples here will follow the Text Editor + Terminal workflow in the Getting Started guide (Chapter 0). In most cases, Python code will be displayed in one block and the following block will show the terminal output of that code. Python code will be color-coded and terminal output will be all black text except for the terminal command to run the file: `$> python file.py`. Some Python files and their outputs will be split between multiple listings, but the file name will always be mentioned explicitly.

If you are familiar with some other languages, you may notice that Python is not compiled. Instead Python is *parsed*, meaning it is read and executed consecutively as the file is read. If what that means is unclear: don't fret. It will make sense as we dive deeper in further chapters. We will begin with the apparently mandatory beginner exercise: "Hello World".

1.1 Hello World

As it is the unwritten law that all programming instruction begin with "Hello World!", we will implement it in Python to get started. This does have utility in demonstrating the simplicity of Python and the nature of it being a parsed language. Create a file named `helloWorld.py`, and all that we have to do is tell Python to print our greeting to the world with the one-liner in Listing 1.

```
1   print("Hello World!")
```

Listing 1: Classic "Hello World!" code

Running the `helloWorld.py` program from a terminal will ask the Python parser to look at this file, and process the commands as they are read. Anything that is *top level*, meaning anything not contained in a function, class, etc. will be executed as it is read and translated. This will yield the output in the terminal as we see in Listing 2.

```
1  $> python helloWorld.py
2  Hello World!
```

Listing 2: Running helloWorld.py from a shell

Chapter 2

Variables, Operations, and Basic Output

2.1 Variables, Operations, and Types

2.1.1 Variables

Variables in Python are *assigned* using the = operator. The order of input is very important. The variable name, which can be any combination of letters, numbers, hyphens, and underscores - so long as the name begins with a letter. The case of the letters can be as you wish however be aware that Python variables are case sensitive, i.e. m and M are different variable names. When *declaring* a variable (writing and assigning it a value for the first time), you do not need to specify what type of data it will hold. If you are familiar with a language like C, this will be quite different. Variables in Python are dynamic, so you can change them in just about any way you want. In listing 3, we assign integers 6 and 3 to the variables x and y.

```
1   x = 6
2   y = 3
```

Listing 3: Assigning values to the variables x and y.

Style Notes: Variable Names

When naming variables there are a few things to consider. Most important is to use names that are concise but still descriptive enough for you to easily understand your code and remember variables. This extends to code that may be shared; if someone else is working with your code and has to figure out what var1 and abcd are for it will be very difficult. This doesn't mean that somewhat long names are always bad, but it is up to you to decide what works best for each case. For variables that have more than one word as part of the name there are a few styles that are common. The style that I will use throughout these documents and examples is camelCase. Camel case begins with a lower case letter and each new word has its first letter capitalized. Some examples are in Listing 4.

```
1   initVelocity = 6
2   spatialFrequency = 3
3   maxFreqX = 10
```

Listing 4: Assigning values to the variables with multi-word names using camelCase.

Another common style is `snake_case`. This style uses an underscore between words. Which style you use is up to you (or your group or employer sometimes), but definitely don't just string words together in variable names without using a style that breaks apart words visually. Other style guidelines will be presented throughout these documents and examples. Example: constant variables that shouldn't be changed are often named in all capital letters. Maintaining a clean and readable style is essential in group work as well as solo coding so that you can read and modify your code again when you haven't looked at it in a while.

2.1.2 Basic Operations

Now that you are familiar with the basis of starting a Python script and displaying output, it is time to cover the basic mathematical operations you can use in Python to do thing more interesting than printing out a nice message. These are all pretty self explanatory, but keep in mind that the syntax for exponents differs from some other programming languages:

- Addition: `+`
- Subtraction: `–`
- Multiplication: `*`
- Division: `/`
- Exponent: `**`
- Modulus (remainder):   `%`
- Floor Division: `//`

Example usage of these operators is shown in Listings 5-8. Everything is straightforward in these examples, but it is important to take note of when assignment (saving the result of an operation) occurs. When performing operations with variables, no changes to any variables occurs *unless* a variable is explicitly changed with the *assignment* operator `=`. This is seen in these example listings when assigning the results to `z`.

```
1  print("Defining our variables:")
2  x = 6
3  y = 4
4  print(x, y)
5  print("x + y =", x + y)
6
7  z = x + y
8  print("z = x + y =", z)
9
10 z = x - 1
11 print("z =", z)
```

```
1  $> python opsIntro.py
2  Defining our variables:
3  6 4
4  x + y = 10
5  z = x + y = 10
6  z = 5
```

Listing 5: Lines 1-5 of `opsIntro.py` demonstrating variable assignment and using the + and -
operators. It is important to note that the result of `x + y` within `print` is not stored. The
result is printed and is not saved afterwards. `x` and `y` will still be `6` and `4` respectively. The
output of this section is shown in the second frame.

```
12
13 print("x * 2 =", x * 2)
14 z = x * y
15 print("z =", z)
16
17 print("x / 2 =", x / 2)
18 print("x / y =", x, "/", y, "=", x / y)
19
```

```
7  x * 2 = 12
8  z = 24
9  x / 2 = 3.0
10 x / y = 6 / 4 = 1.5
```

Listing 6: Lines 12-19 of `opsIntro.py` demonstrating use of the * and / operators. Note that in
the output in the second frame, that after division the result is displayed with a decimal point.
Since division may not always yield a whole integer, the *type* is changed to *float*. Types are
discussed further in Section 2.1.3.

```
20  print("store x / y in z..")
21  z = x / y
22  print("z =", z)
23  print("z + 2 =", z + 2)
24  print("z * 2 =", z * 2)
25
```

```
11  store x / y in z..
12  z = 1.5
13  z + 2 = 3.5
14  z * 2 = 3.0
15
```

Listing 7: Lines 20-25 of `opsIntro.py` and its output. This section demonstrates that any further operations on `z` after division changed its type retains that type, even if the result is an integer.

```
26  print("\nx squared is x**2 =", x**2)
27  print("the remainder of x / y is: x % y =", x % y)
28  print("16 / x rounded down is: 16 // x =", 16 // x)
29  print("x(y + z) =", x * (y + z))
```

```
16  x squared is x**2 = 36
17  the remainder of x / y is: x % y = 2
18  16 / x rounded down is: 16 // x = 2
19  x(y + z) = 33.0
```

Listing 8: Lines 26-29 of `opsIntro.py` and its output. This section provides an example of each of the common operators beyond simple arithmetic. Note that both the `%` and `//` operators yield integer output.

Key Point - Operators

- Variables don't change unless explicitly changed with the `=` assignment operator.

- No operations are implied. A variable next to parentheses will cause an error unless the `*` operator is used (line 29 of Listing 8).

- Operations can change data types of variables (line 17 of Listing 6 when using `/`). More on data types can be found in Section 2.1.3.

- Order of operations is as follows (first to last). Operators of the same group work from left to right in the code, except for `**` which reads from right to left.

    `**`

    `*, /, //, %`

    `+, -`

Note that this operator list does not cover every Python operator. Further operators will be introduced in their applicable topics.

2.1.3   Data Types

Every variable in Python has a *type*. The type of a variable is key in terms of what can be done with it, and how it behaves differently from other types under the same circumstances. For example if I have a variable `x = "s"` and I say `x + 1`, it doesn't work. `"s"` isn't a number and we don't have a defined way to add a number to it. Type conversion is possible when needed (and when it makes sense to do so). First, here are the basic data types to get started with (name, keyword, examples):

- Integer: `int`   Ex: 1, 10, 121321, -982

- Float: `float`   Ex: 1.0, 10000.03, 12.687, -1.982

- Complex: `complex`   Ex: 1 + 10j, 10.0 - 0.2j, 0.004j, -982 + 1.923j

- String: `str`   Ex: "words", "example words", "I miss Vine"

- Boolean: `bool`   Ex: True, False (Capitalization is *required*!)

Note that for complex numbers only `j` will work and not `i`. Examples demonstrating how to work with complex numbers properly in Python will follow in later chapters. Type conversions can be achieved when needed by placing the variable within parentheses after the desired type. For example if a string contains only an integer, it can be converted to an integer type before performing a mathematical operation. An example shown in Listing 9.

```python
textVar = "100"
numVar = 80
result = numVar + int(textVar)
print(result)
```

Listing 9: Converting a `string` type variable to an `int` type for addition. The script prints 180 as the result. Note that if textVar were `"100.5"` or even `"100.0"` there would be an error as Python sees that it is a `float`.
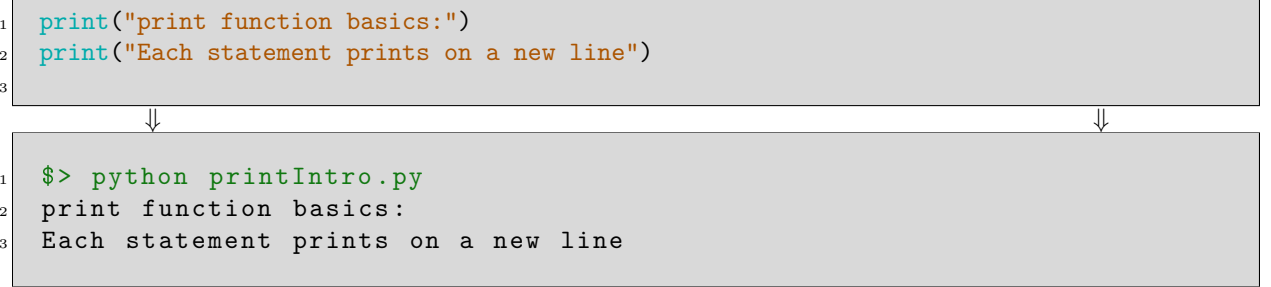
2.2   Printing Basics

In most cases in which you're running your program there will be some result that you'll be wanting to display. There are many forms to do so such as printing, plotting, output to a file, etc. We will start with the simplest - printing. We've already printed the classic greeting using `print`, but there are a few more

features beyond that single message available. The following breaks down the example file `printIntro.py` to exhibit these features. This is a good start, and our terminal output will be further extended in a later section about formatting.

2.2.1   First Features of `print` and Strings

As we have seen in examples using `print`, every individual `print` statement is on its own line in the output. Do also consider that terminals don't always word-wrap, and that even if they do it often gets difficult to read. Keeping statements that fit within a standard sized terminal (usually about 80 characters) is good practice for usability, and multiple print statements is one way to achieve this, seen in the short demo of Listing 10.

```
1  print("print function basics:")
2  print("Each statement prints on a new line")
3
```

⇓   ⇓

```
1  $> python printIntro.py
2  print function basics:
3  Each statement prints on a new line
```

Listing 10: Lines 1-3 of `printIntro.py`. The call to run the script and the output of this section are shown in the second frame. Each print statement gets printed on its own line in the output.

The `end` parameter

The first useful feature is the ability to set the ending for a print statement using a function *parameter* (sometimes also called an *argument*). This is done within the call to `print` (i.e. within its parentheses) by setting the `end` parameter. This parameter is by default a special character that tells the terminal to move to the next line (like hitting the "enter" key in a text editor). Function parameters that have default values that can be manually changed are called *optional parameters*. Setting the `end` parameter manually by using `end="..."` within `print` overrides this newline default, allowing for customization of output. This special newline character is discussed within this chapter in Section 2.3. Looking at Listing 11, it is shown that setting the `end` parameter to `""` leaves nothing as the ending, so the cursor that Python is using to output to the terminal stays where it is until the next print statement. Note that it is required to put the empty quotations `end=""` and doing something like `end=` will not work as the `=` operator will always operate on the next character - resulting in an error.

In some cases you may want to set the ending to something more interesting than nothing. This can be done by putting anything you'd like *within* the quotes on the `end` parameter, as shown in Listing 11. As you read through the example code, notice the inconvenient use of two empty `print` statements on lines 15 and 16 of Listing 11. A good rule of thumb is if portions of code feel repetitive to write: there is likely a better way to write it. We'll see a better solution than these two empty `print`s shortly in Section 2.3.

Multiple Strings in `print`

Multiple strings (or variables) can be used in a `print` by separating them by a comma, allowing for a theoretically infinite number of statements input into one `print` - though more than a few becomes impractical. When using `string`s as we are, you can also do what's called *concatenation* - or combining two strings into one longer string. This can be done either with a juxtaposition of the two strings in quotes, or by using the `+` operator between them. Using `+` is usually easier to spot and read on the development side, so that method is recommended. All of this is shown in Listing 12.

```python
2   print("Each statement prints on a new line")
3
4   print("unless I tell it not to with the end= ", end="")
5   print("parameter")
6   print("- Because by default end is set to a newline")
7
8   print()
9   print("an empty print still uses the default end parameter")
10  print()
11
12  print("I can end with spaces", end="   ")
13  print("or dashes", end="---")
14  print("or anything I need to", end="!!!!!")
15  print()
16  print()
17
18  print("I can print two things", "together with a comma")
```

```
3   Each statement prints on a new line
4   unless I tell it not to with the end= parameter
5   - Because by default end is set to a newline
6
7   an empty print still uses the default end parameter
8
9   I can end with spaces   or dashes---or anything I need to!!!!!
10
```

Listing 11: Lines 2-18 of `printIntro.py` and its output. Here the effects of changing the `end` optional argument are shown. Note that empty `print` statements here print a blank line in the output.

The `sep` parameter

The other major feature for `print` customization is the `sep` parameter. Setting this parameter will change what is automatically printed between comma separated values, which is by default a space: `sep=" "`. This can be adjusted to anything including empty quotes `sep=""` or a variable which we saw in Section 2.1. A short example is shown in Listing 13.

Lastly for the `printIntro.py` example, we address an issue regarding quotes. Strings can be defined either with single quotes: `'this is a string'` or double quotes `"this is also a string"`. Which one to use is a matter of preference, but it is important to note that if you're using one then you can only use the other within the string. An example of this is shown in Listing 14, but a workaround is addressed in the following Section.

```
18  print("I can print two things", "together with a comma")
19  print("- and if they are strings I can " "omit the comma")
20  print("- or I can use " + "the + operator")
21  print("-- but not with variables! we saw that in the variables section.")
22  print()
23
24  print("If I want something specific between inputs, I can do that:")
```

⇓   ⇓

```
10
11  I can print two things together with a comma
12  - and if they are strings I can omit the comma
13  - or I can use the + operator
14  -- but not with variables! we saw that in the variables section.
15
```

Listing 12: Lines 18-24 of `printIntro.py` and its output. This shows various methods of *concatenation* with strings. Note that with variables of other types we can only use the comma separator - `print` then handles the conversion to the `string` type.

```
24  print("If I want something specific between inputs, I can do that:")
25  print("using the", "sep", "parameter", sep="--")
26  print("But I have to use commas" " to do so", sep="dskjfas;dk", end="\n\n")
27
28  print("As long as I use quotes for strings, I can use an apostrophe ' ")
```

⇓   ⇓

```
15
16  If I want something specific between inputs, I can do that:
17  using the--sep--parameter
18  But I have to use commas to do so
```

Listing 13: Lines 18-24 of `printIntro.py` and its output. This section demonstrates changing the `sep` optional argument. Note that this parameter only changes output between comma separated values in `print`.

```
28  print("As long as I use quotes for strings, I can use an apostrophe ' ")
29  print('but if I use single quotes, I cannot, but I can use "quotes"')
30  print("- we'll see how to get around this later")
```

⇓   ⇓

```
18  But I have to use commas to do so
19
20  As long as I use quotes for strings, I can use an apostrophe '
21  but if I use single quotes, I cannot, but I can use "quotes"
```

Listing 14: Lines 28-30 of `printIntro.py` and its output. This section demonstrates that `string`s can be defined with either double or single quotes, and that determines which can be used within the string. A solution to this issue follows in the next subsection.

Key Points - Printing Basics

- Each default call to `print` will print the input on one line

- `print` has a few optional parameters:

  - `end` sets the string printed after printing the input(s)
    -default: new line

  - `sep` sets the string printed between comma separated inputs
    -default: space

- Multiple strings (or variables) can be printed in one `print` statement.

2.3 Escape Characters

As you may have noticed in portions of 2.2, some code was repetitive. Using `sep` and `end` add helpful features but still leave `print` feeling a little stiff in its usage. This is where escape characters come in. These are special characters that you may know as "invisible characters" in a word processor, as well as a way to "escape" out of the function that some characters normally have in the Python language. Look at how we can make use of quotes within a string defined by quotes in Listing 15; this is a solution to the problem we saw in Listing 14.

```python
print("Python has \"escape characters\"")
print("- which were used to print quotes in a string defined with quotes")

print()
```

```
$> python printIntroEscapes.py
Python has "escape characters"
- which were used to print quotes in a string defined with quotes

```

Listing 15: Lines 1-4 of `printIntroEscapes.py` and its output. In this section, the backslash is used to "escape" the usage of quotes. This allows them to be printed without breaking the quotes used to define the `string`.

Escape characters can be used in any string, even those that you are not using immediately in a `print`. When they are in a `print` though, a new line escape `\n` can be used - saving you from writing an empty print statement to write blank lines. The use of this new line special character is shown in Listing 16. One thing to be careful of here is typing in an extra space. If you were to type `end="\n "`, then at the beginning of your new line, there would be a space as well.

As just mentioned, escape characters can be in *any* string, this includes the strings put into the `end` and `sep` parameters. This adds another degree of customization and flexibility. Basic examples are in Listings 17.

To summarize escape characters, the most commonly used ones are new line, tab, backslash, single quote, and double quote. These are mentioned in `print`s in Listing 18. This example does not encompass all escape characters - further ones are typically in more advanced applications but a curious beginner can find many examples in a web search.

```python
4  print()
5  print("Instead of empty prints, I can print newlines\n - with \\n")
6
7  print("\nI can put them\nanywhere\nin the string\n")
8
9  print("notice that \\ only affects the next character")
10
11 print("I can even change the end with these", end="\n\n")
```

```
4
5  Instead of empty prints, I can print newlines
6   - with \n
7
8  I can put them
9  anywhere
10 in the string
11
12 notice that \ only affects the next character
```

Listing 16: Lines 4-11 of `printIntroEscapes.py` and its output. This section demonstrates the usage of the newline character `\n`, and that the backslash only applies to the adjacent character.

```python
11 print("I can even change the end with these", end="\n\n")
12 print("or separate", "inputs with", "tabs", sep="\t")
13 print()
14
15 print("Adjust both \"sep\"", "and", "\"end\" as you wish", sep="\t", end="\n\n")
16
17 print("Common escape characters:")
```

```
13 I can even change the end with these
14
15 or separate     inputs with     tabs
16
17 Adjust both "sep"      and      "end" as you wish
18
```

Listing 17: Lines 11-17 of `printIntroEscapes.py` and its output. The adjustment of optional parameters `end` and `sep` with escape characters is demonstrated here.

```
17  print("Common escape characters:")
18  print("\\n \\t \\\ \\' \\\"", end="\n\n")
19  print("Other more specialized ones also exist")
```

```
18
19  Common escape characters:
20  \n \t \\ \' \"
21
22  Other more specialized ones also exist
```

Listing 18: The final lines of `printIntroEscapes.py` and its output, summarizing the most often used escape characters. Note that to print a backslash, it needs to be escaped: \\.

Key Points - Escape Characters

- Escape characters are special characters that can be in strings
- These characters are "escaped" using the backslash \
- Common escape characters:
  - New line: `\n`
  - Tab: `\t`
  - Backslash: `\\`
  - Single quote: `\'`
  - Double quote: `\"`

2.4 Full Code Listings

Chapter 1

```
1  print("Hello World!")
```

Listing 19: The full content of `helloWorld.py` used in Section 1.1

```
1  $> python helloWorld.py
2  Hello World!
```

Listing 20: The full output of `helloWorld.py`

Chapter 2

```python
print("Defining our variables:")
x = 6
y = 4
print(x, y)
print("x + y =", x + y)

z = x + y
print("z = x + y =", z)

z = x - 1
print("z =", z)

print("x * 2 =", x * 2)
z = x * y
print("z =", z)

print("x / 2 =", x / 2)
print("x / y =", x, "/", y, "=", x / y)

print("store x / y in z..")
z = x / y
print("z =", z)
print("z + 2 =", z + 2)
print("z * 2 =", z * 2)

print("\nx squared is x**2 =", x**2)
print("the remainder of x / y is: x % y =", x % y)
print("16 / x rounded down is: 16 // x =", 16 // x)
print("x(y + z) =", x * (y + z))
```

Listing 21: The full content of `opsIntro.py` used in Section 2.1.2

```
1  $> python opsIntro.py
2  Defining our variables:
3  6 4
4  x + y = 10
5  z = x + y = 10
6  z = 5
7  x * 2 = 12
8  z = 24
9  x / 2 = 3.0
10 x / y = 6 / 4 = 1.5
11 store x / y in z..
12 z = 1.5
13 z + 2 = 3.5
14 z * 2 = 3.0
15
16 x squared is x**2 = 36
17 the remainder of x / y is: x % y = 2
18 16 / x rounded down is: 16 // x = 2
19 x(y + z) = 33.0
```

Listing 22: The full output of `opsIntro.py`

```python
print("print function basics:")
print("Each statement prints on a new line")

print("unless I tell it not to with the end= ", end="")
print("parameter")
print("- Because by default end is set to a newline")

print()
print("an empty print still uses the default end parameter")
print()

print("I can end with spaces", end="   ")
print("or dashes", end="---")
print("or anything I need to", end="!!!!!")
print()
print()

print("I can print two things", "together with a comma")
print("- and if they are strings I can " "omit the comma")
print("- or I can use " + "the + operator")
print("-- but not with variables! we saw that in the variables section.")
print()

print("If I want something specific between inputs, I can do that:")
print("using the", "sep", "parameter", sep="--")
print("But I have to use commas" " to do so", sep="dskjfas;dk", end="\n\n")

print("As long as I use quotes for strings, I can use an apostrophe ' ")
print('but if I use single quotes, I cannot, but I can use "quotes"')
print("- we'll see how to get around this later")
```

Listing 23: The full content of `printIntro.py` used in Section 2.2

```
1  $> python printIntro.py
2  print function basics:
3  Each statement prints on a new line
4  unless I tell it not to with the end= parameter
5  - Because by default end is set to a newline
6
7  an empty print still uses the default end parameter
8
9  I can end with spaces   or dashes---or anything I need to!!!!!
10
11 I can print two things together with a comma
12 - and if they are strings I can omit the comma
13 - or I can use the + operator
14 -- but not with variables! we saw that in the variables section.
15
16 If I want something specific between inputs, I can do that:
17 using the--sep--parameter
18 But I have to use commas to do so
19
20 As long as I use quotes for strings, I can use an apostrophe '
21 but if I use single quotes, I cannot, but I can use "quotes"
22 - we'll see how to get around this later
```

Listing 24: The full output of `printIntro.py`

```
1  print("Python has \"escape characters\"")
2  print("- which were used to print quotes in a string defined with quotes")
3
4  print()
5  print("Instead of empty prints, I can print newlines\n - with \\n")
6
7  print("\nI can put them\nanywhere\nin the string\n")
8
9  print("notice that \\ only affects the next character")
10
11 print("I can even change the end with these", end="\n\n")
12 print("or separate", "inputs with", "tabs", sep="\t")
13 print()
14
15 print("Adjust both \"sep\"", "and", "\"end\" as you wish", sep="\t", end="\n\n")
16
17 print("Common escape characters:")
18 print("\\n \\t \\\ \\' \\\"", end="\n\n")
19 print("Other more specialized ones also exist")
```

Listing 25: The full content of `printIntroEscapes.py` used in Section 2.3

```
$> python printIntroEscapes.py
Python has "escape characters"
- which were used to print quotes in a string defined with quotes

Instead of empty prints, I can print newlines
 - with \n

I can put them
anywhere
in the string

notice that \ only affects the next character
I can even change the end with these

or separate      inputs with      tabs

Adjust both "sep"       and      "end" as you wish

Common escape characters:
\n \t \\ \' \"

Other more specialized ones also exist
```

Listing 26: The full output of printIntroEscapes.py

Index