Basics - Introduction

Daniel Scarbrough

February 13, 2021

Contents

Chapter 1

Introduction

Many people who are either experienced in a language such as

Python however operates through a *parser*, meaning that

1.1 Hello World

As it is the unwritten law that all programming instruction begin with "Hello World!", we will implement it in Python. 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

1.1.1 Defining Main

While one can write large Python scripts entirely in the top level, it is usually not a good idea to do so for reasons that may not seem clear now, but will as your projects grow and become more complex. This is where the concept of "style" begins to come into play, as many who are new to Python and programming in general will argue that the code does what it's supposed to regardless of organization and style. While this is true, it is important to consider the experience that you as the developer will have as things grow in complexity.

Keeping code modular and building portions one chunk at a time while being able to test individual pieces becomes an essential feature to you as the author of the code. Keeping with these style recommendations is

an investment in your future productivity, and more importantly, your sanity.

With this in mind, let's modify our "Hello World!" code to include a `main` function. While this seems like a lot of added complexity just to print "Hello World!", this structure is the foundation for our modular coding style whose purpose will become clearer in later sections. We will define a `main` function where the instructions for Python will start. Anything that is contained within `main` ***must*** be indented *once* past `def`. Python uses tabs to understand when blocks such as functions end, and the parser will complain immediately if it sees an erroneous tab.

We will discuss functions more in a later section, but for now it is ok to copy the structure from the exampe code. Looking at Listing 3, after `main` is defined it is then *called*. For example if line 4 is omitted (try it), then when the script is run - nothing happens! You've told Python what `main` is but you haven't asked it to use it. This is because the code in `main` are not *top-level*. So thinking of functions as one would in mathematics, we can define an equation for a line $f(x) = mx + b$; but without asking for f at some point- i.e. $f(4)$ nothing will "happen".

Another key is that the open and close parentheses *must* be included even if there are no arguments to be passed through (in the math example $f() = 4$). Meaning the same instructions will be executed every time this function is called. Without the parentheses Python reads `main` as a reference to the function itself - but not an instruction to execute the function. Running the code in Listing 3 will give the same output as in Listing 1, but now the structure is being improved for the developer as well as the user.

```
1 def main():
2     print("hello world")
3
4 main()
```

Listing 3: A modification to "Hello World" to define a main path of execution

Checking the Entry Point

There is one more modification which may seem unnecessary at the moment, but it is worth making a habit of now before the complexity of these Python scripts increases. A major feature in Python is writing multiple *modules* which you can think of as different toolsets that you bring together to solve a problem.

When writing new modules you will want to test them to make sure they do what you expect. You may also use functions in one project which you later realize will be useful in another. When you import the functions from the original project, you won't want to run all the code specific to that project, but you also shouldn't have to make changes that make that project inoperable either. If you were working on a home improvement project for example: you wouldn't want the drill to start spinning wildly once you opened the toolbox, but your solution shouldn't be to throw away the battery.

This is where the keyword `__name__` comes into play. *This is not a variable you can set.* This is set by the Python interpreter and in simple terms, it's Python setting the hierarchy between modules. The file that you give Python in the terminal when you say `$> python fileName.py` is read as a module with the `__name__` set to `__main__`. This is the *entry point* for your program, and it can be thought of as the trunk of a tree for which other modules are its branches. Checking for this is done as in Listing 4. For now it should be used as a starting template; the details around using `if` statements will come later.

NOTE: There are two equals signs in the `if` statement. This is very important and will be discussed later.

The output from running `helloWorldMain.py` is shown in Listing 5, which confirms the value of `__name__`. The importance of this method will be made clear in a later section when we make use of modules within our main file. For now the structure in Listing 4 can be used as a template to start new Python files.

```
def main():
    print("hello world")
    print("I am in", __name__)

if __name__ == "__main__":
    main()
```

Listing 4: A modification to "Hello World" to ensure `main()` is only run if the file `helloWorldMain.py` is the entry point for execution. A `print` statement is added so that we can see what `__main__` is.

```
$> python helloWorldMain.py
Hello World!
I am in __main__
```

Listing 5: Running helloWorldMain.py now also prints the value of `__name__`

Key Points - Defining Main

- `print` can print strings to the terminal
  - `print("enter text to print here")`
- Each Python file can be used as a module
  - Many modules can be used together in another
- Keeping running/testing code in a function like `main` keeps modules "modular"
- Checking if a module is named `__main__` to run certain functions is good practice for future projects

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 are dynamic, so you can change them in just about any way you want. In listing 15, we assign integers 6 and 3 to the variables `x` and `y`.

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

Listing 6: 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 16.

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

Listing 7: 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: `//`

```
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 8: 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.

Key Point - Operators

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

```
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 9

```
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 10

  - This is first shown in lines 9-11 of Listing **??**, as well as in many of the other print statements.

- No operations are implied. A variable next to parentheses will cause an error unless the `*` operator is used (line 50 of Listing **??**.
- Operations can change data types of variables (line 36 of Listing **??** when `x / y` is not an integer. More on data types in Section 2.3.3.
- 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.

```
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))
```

⇓ ⇓

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

Listing 11

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. 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. Example shown in Listing 21.

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

Listing 12: 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 the 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 saw in Listing 4, 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 anywhere from 80-120 characters) is good practice for usability, and multiple print statements is one way to achieve this, seen in the short demo of Listings **??** and **??**

```
1 print("print function basics:")
2 print("Each statement prints on a new line")
3
4 print("unless I tell it not to with the end= ", end="")
```

⇓ ⇓

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

Listing 13

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). Setting it 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 2.2. It is worth noting that this is an *optional parameter*, as its default is always a new line if `end` is not specified. Looking at Listings **??** and **??**, 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 Listings **??** and **??**. You might ask why one would put these characters in the end parameter rather than just the quotes of the string to be printed - this is a good question and it will become clear when we introduce variables in Section 2.3. For now keep this feature in mind, and notice the inconvenient use of two empty `print` statements on lines `16` and `17` of Listing **??**. 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 2.2.

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 Listings **??** and **??**

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

```
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")
```

⇓ ⇓

```
Each statement prints on a new line
unless I tell it not to with the end= parameter
- Because by default end is set to a newline

an empty print still uses the default end parameter

I can end with spaces   or dashes---or anything I need to!!!!!

```

Listing 14

can be adjusted to anything including empty quotes `sep=""` or a variable which we will see in Section 2.3. A short example is shown in Listings **??** and **??**

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 Listings **??** and **??**, but a workaround is addressed in the following Section.

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.

```
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'll see that later")
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'll see that later
15
```

Listing 15

```
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 ' ")
```

⇓ ⇓

```
14 -- but not with variables! we'll see that later
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 16

2.3 Escape Characters

As you may have noticed in portions of 2.1, 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 Listings **??** and **??**; this is a solution to the problem we saw in Listing **??**.

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 Listings **??** and **??**.

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 **??** and **??**

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 Listings **??** and **??**. This example does not encompass

```
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 17

```
1 print("Python has \"escape characters\"")
2 print("- which were used to print quotes in a string defined with quotes")
3
4 print()
```

⇓ ⇓

```
1 $> python printIntroEscapes.py
2 Python has "escape characters"
3 - which were used to print quotes in a string defined with quotes
4
```

Listing 18

all escape characters - further ones are typically in more advanced applications but a curious beginner can find many examples in a web search.

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

```
print()
print("Instead of empty prints, I can print newlines\n - with \\n")

print("\nI can put them\nanywhere\nin the string\n")

print("notice that \\ only affects the next character")

print("I can even change the end with these", end="\n\n")
```

```


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
```

Listing 19

```
print("I can even change the end with these", end="\n\n")
print("or separate", "inputs with", "tabs", sep="\t")
print()

print("Adjust both \"sep\"", "and", "\"end\" as you wish", sep="\t", end="\n\n")

print("Common escape characters:")
```

```
I can even change the end with these

or separate	inputs with	tabs

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

```

Listing 20

```
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 21

Index