Contents

1 Python comments

Comments in programming are not meant to be read and interpreted by the computer. They function more for the reader of the code, to provide to them some context, information, background, or anything that the code author wished to convey.

In Python, *single-line comments* are denoted with the # symbol. When Python encounters this symbol, it ignores everything to the right of it. *Multi-line comments* are delimited by a matching pair of three quote marks, either `'''` or `"""`, and ignore everything between the pair of quote marks.

```python
''' An example of a multiline comment
"Programs must be written for people to read, and only incidentally for
    machines to execute."
- Harold Abelson & Gerald Jay Sussman
'''
module = "PS0001"     # Introduction to Computational Thinking
lab = 3               # Week 3
author = "Caleb Tay"  # :)
print("Hello, and welcome to", module, "lab", lab)
#print("I am commented out. I don't run.")
print("This lab was written and developed by", author)
```

2 Data types

A computer only understands numbers, specifically, binary. So how does Python (and your computer), represent anything more than just a number? How do we express strings of characters, like `Python is cool` to a computer, and how does it convert between that string and a sequence of 1s and 0s?

A computer differentiates and understands what something is in memory by associating it with a particular **data type**. There are many data types out there, but this week, we will learn about 3 of them: the integer, the float, and the string.

2.1 Integers (`int`)

The integers are the whole numbers, positive, negative, and 0. You may know them in math as $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots,\}$. Realistically, a computer can't count infinitely - there are maximum values that a computer can represent, past which the computer would interpret as $\pm\infty$. For now, this is sufficiently accurate for our purposes.

You've been handling integers in the previous lab, adding, subtracting, multiply and dividing them, and it works exactly as in math. Adding, subtracting, and multiplying two **`int`**s gives you another **`int`**. Dividing two **`int`**s in Python gives you a **`float`**, providing some level of decimal accuracy.

```python
print("12+34 =", 12+34) # an int
print("56-78 =", 56-78) # an int
print("12*89 =", 12*89) # an int
print("34/67 =", 34/67) # a float
```

2.2 Floats (`float`)

Floats are what in math we may refer to as the rational numbers $\mathbb{Q}$ or the real numbers $\mathbb{R}$. Again, not quite accurate, since computers cannot represent infinitely many decimal digits (e.g. as in π or $\sqrt{2}$), but this definition suffices for our purposes. A float in Python is essentially any number which has a decimal point (e.g. `2.718`, `2.0`).

Adding, subtracting, multiplying, and dividing two floats gives you another float. You can also try mixing these operations with integers.

2.3 Strings (`str`)

Strings are sequences of characters, and are enclosed by quote marks, single or double. We've been using them, asking the `print()` function to display them in your console. Multi-line strings can also be defined, and use the same syntax as multi-line comments.

You can perform some operations on strings, which you will experiment with in the exercises.

```
first = 'this is a stirng'
second = "this is also a string"
third = '''i am
a multi-line
string'''
print(third)
```

2.4 The `type()` function

There are more types than just integers, floats, and strings out there. You will learn of a few more types in the coming weeks, the most important of which will be lists. One thing in common, however, is that you can use the `type()` function to check and see what type a value/variable is.

```
print(type(123))   # int
print(type(123.0)) # float
print(type("123")) # str (a string of numbers)
```

3 Implementing user interactivity - the `input()` function

The lab and lab exercises thus far has involved only typing code into a file editor - so how do you introduce interactivity into your program? The most basic way of taking *input* from a user in Python is through the `input()` function.

While the **`print()`** function displays something to the user, the `input()` function will wait for the user to input something, followed by an enter, before the program will continue:

```
print("Hello!")
input("press a key.")
print("World!")
```

Of course, it's not just used to hold up a program from executing - anything that the user inputs is returned to Python, as a string of characters. You can pass `input()` one parameter, which can act as a prompt:

```python
your_name = input("What is your name? ")
print("Hello, it is nice to meet you,", your_name)
```

The below program will raise an error. Why?

```python
your_age = input("What is your age? ")
print("You will be", your_age + 2, "years old in 2 years time.")
```

`your_age` is a string, and 2 is an integer. Python is able to do some operations between some types. In this case, Python does not know how to add a string and an integer together. Python will crash for *any* operation it is unaware of how to perform. You will experiment with some of these in the exercises.

To fix the issue, we must convert `your_age` into a data type that allows us to do what we want to do, which is to add 2 to it. In Lab 1's hard exercise, the `int()` function was introduced, which removes the decimal/fractional portion of a number. It can also be used to convert a string (of numbers) into an integer. Before running the below code, what do you expect the output to be?

```python
three = '3'
print("3 times 3 is", three * 3)
three = int(three)
print("3 times 3 is", three * 3)
```

In the same manner, we can convert a user's input taken via the `input()` function, which is always a string, into an integer using `int()`.

```python
your_age = int(input("What is your age? "))
print("You will be", your_age + 2, "years old in 2 years time.")
```

Converting strings into floats works much the same with with `float()` - again, we encourage you to experiment with these functions. One peculiarity with the `int()` function is that it is unable to convert strings with decimals into integers, and will raise an error.

```python
float("3") + 2
int("3") + 2
float("3.14159") + 2
int("3.14159") + 2
```

You may have wondered: what happens if I ask my user for a number as an input, and convert it, but they entered some letters (e.g. `'abc'`)? For the time being, until much later, we will assume that the user is *well-behaved*, and not trying to break your program. You will learn how to deal with bad input like this in a future lab, where we learn more about errors, exceptions, and how to deal with them.

4 Errors

Throughout this course, you will encounter many errors when trying to run your code. Such errors interrupt the execution of your code, and cause them to crash, without completing the execution. We will just briefly mention the two most common that you will likely encounter, and how you can diagnose and fix these errors in your code.

You can run each cell for yourself to see the error that occurs.

4.1 `NameError`

In the error message below, there are only two key things to pay attention to: the **error type**, and **where it occurred**.

The error type is easy to see - it is the very last line that is printed, and it indicates that it is a `NameError`, with a helpful description of what variable is not defined.

As for where the error occurred, it is right above the error type, in the **traceback**. Without going into details, the traceback gives us a *trace* of how the error occurred. This traceback is helpful when you have much more code to run, and need to find exactly where it occurred, and what led up to the error occurring. In this case, the error occurred in the same cell, at line 2.

In Spyder, you can click on the line number to navigate to the relevant line that caused the error in the file editor. You should fix this issue while you're here!

```python
x = 12
print(a)
```

4.2 `SyntaxError`

This will likely be the most common error you will encounter when starting off in Python, and it comes in many forms. Syntax errors are caught **before any code is run**, when Python checks if it can understand the given code. If Python cannot understand the code (i.e. there is a `SyntaxError`), it will not run the code at all. Each example below is a different type of syntax error.

4.2.1 Use of keywords as variable names

While this error points out that the = symbol causes the error, it notably does not indicate that `import` is a keyword that cannot be used as a variable. A simple fix would be to prefix the name with an underscore character, though you shouldn't be facing this particular scenario often. Ideally, you should be naming your variables something different.

```python
import = 0
```

4.2.2 Improper indentation

In Python, indentation (the amount of space at the beginning of a line) is incredibly important. You will see why in a few weeks, but **for now**, make sure that every Python statement begins *without* any spaces at the front.

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

4.2.3 Unmatched brackets

This one is simple - just as in math, when your brackets aren't matched, you will have a syntax error.

While it can indicate that there is an unmatched `)`, when you have multiple layers of bracketing, removing the indicated bracket may change the meaning of your code significantly. For example, you may see this happen with a longer, complicated mathematical formula, where removing one bracket may change the order of operations significantly.

Again, there are many more errors than these, and we will revisit a few more errors in a future lab.

```
print())
```