Introductory Computer Science

Week 2 — Programming with Python

Dann Sioson
dj.sioson@alum.utoronto.ca

June 23, 2019

Table of Contents

Table of Contents

Administrative items

- Assignment is up, due a day before Week 6 rendezvous
- This week's exercise is up (along with the marking scheme)
- Extra practice material is available as well
- A test is going to be conducted on week 4
- Itinerary has changed, but still subject to change towards the end

Itinerary

1. ~~Introduction and Git~~
2. Programming with Python*
3. Memory Model and Debugging*
4. While Loops and Recursion
5. Recursion
6. Test day
7. Checkpoint
8. Object Oriented Programming
9. Object Oriented Programming
10. Data Structures?

* = What will be tested
? = Subject to change

Table of Contents

What is an Algorithm?

- A set of instructions
- Broken up atomically
- Put together in sequential order

Wing

- Wing is relatively a nicer text editor than most to introduce people to Python
- However, if you want to, you can use another text editor to your liking
- Benefits of Wing:
  - Dark theme
  - You can click to run a command (as opposed to writing it into shell)
  - Python shell near the bottom
  - Debugger (we'll see this next week)

Python Shell

- The shell waits for an input by showing >>>
- Once you hit Enter, the line(s) is interpreted
- Any values as a result of the line are typically returned to the next line
- EXCEPT FOR PRINT. `print()` shows the string representation of what you put into the parentheses, but `print()` returns a `None` type
- Use the shell for disposable use as nothing inputted is saved (except for commands by pressing up)

.py files

- All your code is saved into a .py file
- ...practically a text file
- For now, the only way you can show your work is by print()
- However, all exercises and assignments will deduct marks if you do keep them upon merging into master

Commenting

Why comment?

- So that you can keep the flow of what you are coding
- If you ever need to go back to old code, you will be able to describe how you did something at a very low level

- Comments in Python are typically denoted in hashtags: #
- A way to practice conciseness
- You will be using a style guide later on, when you start a comment on a line, follow it up with a space

Variables

- Variables are used to point to an abstraction of data (similar to math)
- To follow up with the coming style guide, variables (in this course) will be in `pot_hole_case` and not `CamelCase`

Table of Contents

What are Primitive Types?

- Primitive Types are typically the building-block data types within any language
- Meaning that you can almost expect it in almost every programming language

Integer

- Any number without decimals
- The embodiment of N

Float

- Any number with decimals (excluding imaginary numbers)
- The embodiment of R

Boolean

- Two values
- `True`, and `False`
- Boolean Operators:
    - `and` = if there is at least one `False` statement between two statements, then the whole statement is `False`
    - `or` = if there is at least one `True` statement between two statements, then the whole statement is `True`
    - `not` = The opposite of the statement

Boolean Expressions

- `==` Check if two objects are the same
- `!=` Check if two objects are not the same
- $<$ less than
- $>$ greater than
- $<=$ less than or equal to
- $>=$ greater than or equal to
- *is* : Same memory allocation (we'll get into this next week)

Table of Contents

if statements

- Produce consequences with your code, dictating how your logic will flow
- An if statement begins with `if` followed by (typically) a boolean value, colon, a new line, an indent, then more code code

else-if statements

- If the preceding (else) `if` statement fails, check this condition
- In Python, you make and else if statement by `elif`

else statements

- Typically the last case to say: “okay, if all my cases I did set up failed, do this”
- To make an else statement, it must follow either an if statement (and coding block) or an else if statement (and coding block)
- To make an else statement in Python, you just do `else:` followed by a coding block

Table of Contents

What are Strings?

- Strings, in simplicity, is text
- To make a string in Python, you surround the text with quotes

Concatenating (and other functions)

- Adding onto a String is done by a + sign
- Other functions relating to a string is done by `help(str)`

Indexing

- You can get specific characters from the string by putting a number in a square bracket that follows the string or variable that contains the string
- However, for a lot of programming languages, remember that indexes start at 0, not 1
- You can index a String from 0 to `len(s) - 1`

Substrings

- Substrings are made from strings
- You make substrings in Python by: `my_string[x:y]`
- Where x and y are integers
- x = start and include this character
- y = up to but not including here

[Python Specific] Negative indexing

- Negative indexing gets the character of the string indexed at `len(s) + i` (if it exists)
- Where $i < 0$

Table of Contents

What is a list

- Properties are similar to a string
- However, a list can hold many objects at once
- The types of objects do no matter (but unfortunately, this is Python specific too)

List appending

- There are two ways in which you can append to a list
- Just for a single element it uses the .append(element) method
- If you want to put two lists together, you can use the + sign
- Check `help(list)` for other functionalities with lists

Indexing

- Similar to Strings
- You can get specific elements from a list by putting a number in a square bracket that follows the list or variable containing the list
- Negative indexing is supported in Python as well

Slicing

- Slicing makes a copy of the list (we'll get into how complicated this can be next week)
- Still similar to a string, you can make smaller lists by doing as follows `some_list[x:y]`
- x = Starting and including x
- y = Up to, but not including y

Table of Contents

for-loops

- Typically, to iterate through a list of numbers that have a well defined stop point

```
for number in range(i, j, k*):
    print(number)
```

- i = starting and including
- j = stopping and not including
- k* = optional, skip k many steps for next iteration

Elemental for-loops

- Iterate through an object (that usually holds many objects)

```
my_list = [3, 47, 38]
for element in my_list:
    print(element)
```

while loops

- You can expect a while loop in every programming language (except for functional programming)
- Tends to be harder to grasp, but the week we're going to be tackling while loops on the same week as recursion
- pseudo-code for while loops

```
variable = True
while [condition]:
    # code
    # ...
    variable = ... # evaluate
```

Table of Contents

Why use functions?

- As a programmer, your job is to be as lazy as possible
- That is, you do not repeat code
- Functions helps us not repeat code
- This is where you will be doing most of your code for exercises and the assignment

How functions are structured

They start off with `def` followed by the name of the function, parentheses to indicate parameters, a new line, indent, code, and typically ending off with a `return` statement

```
def
```

How functions are structured

They start off with `def` followed by the name of the function, parentheses to indicate parameters, a new line, indent, code, and typically ending off with a `return` statement

```
def function_name
```

How functions are structured

They start off with `def` followed by the name of the function, parentheses to indicate parameters, a new line, indent, code, and typically ending off with a `return` statement

```
def function_name(parameter_1, parameter_2 ...):
```

How functions are structured

They start off with `def` followed by the name of the function, parentheses to indicate parameters, a new line, indent, code, and typically ending off with a `return` statement

```
def function_name(parameter_1, parameter_2 ...):
    # block of code
```

How functions are structured

They start off with `def` followed by the name of the function, parentheses to indicate parameters, a new line, indent, code, and typically ending off with a `return` statement

```
def function_name(parameter_1, parameter_2 ...):
    # block of code
    return None
```

Design Recipe for functions

- If we stopped here, it's not enough
- A lot of bad Python tutorials stop here
- However, a lot of those bad tutorials do not signify the importance of what you are creating
- So there is a "Design Recipe" for functions

Design Recipe

1. Header (and parameters)
2. Type contract
3. Description
4. Requirements
5. Examples
6. Tests
7. Internal Comments

... and then you can code

Leading example

Create a function that returns the largest adjacent difference from a list of integers

Header

The header is the function name. As a user, if the function name (by itself) does not sound useful to me, then I will not care about it.

Type contract

- Analogous to Mathematical notation

$$f : X \to Y$$

- f represents the function
- X is the set of inputs
- Y is the set of outputs
- Same thing goes for functions

```
(str, int) -> bool
```

Description

- After having interest of the function name, what is required to be inputted, and what is outputted, I need to know exactly what I am getting out of your function.
- Keep it concise. Never put in implementation details as every function to me is magic

Requirements

- The boundaries of inputs for the function
- If there are no boundaries, just say `REQ: None`
- However, if there are boundaries, let me know following `REQ:` so that I know exactly what not to put in to cause an error

Examples

- Show me examples of the function, so that it may be clearer for me to understand what your function does
- Typically mock a Python Shell with >>>
- As a rule of thumb, you conduct tests by: the 0th case, the -1st case, the nth case
- You can check if your tests work as follows:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Internal Comments

- This is more for you than it is for me
- Following the theme of Computer Science, break down the problem into pieces
- To do this in programming, it is to plan ahead before you code
- Going in between languages, code can sometimes be quite repetitive, tedious and memory-oriented. You may not always have time to write all pieces of code
- `System.out.println("hello");`
- It may seem short in Python for now, but as you access other libraries or languages, coding is a long process
- Computer Scientists typically spend about 80% of their time actually planning things

Table of Contents

Tuples

- Tuples are similar to lists
- Instead of a square bracket around them, it's a round bracket (in Python)
- However, tuples have static memory
- Meaning that tuples cannot change values by indexes
- And tuples cannot add or remove elements

Dictionaries/Hashmaps

- In Python, dictionaries are made by using squiggly brackets
- You can instantiate a dictionary as follows

```
my_dict = {key_1: val_1, key_2: val_2...}
```

- Or you can instantiate an empty dictionary by squiggly brackets
- To get a value, you index the dictionary by the key

```
my_dict[key_1]  # this returns val_1
```

- Dictionaries do not have any particular order. However, you can iterate through a dictionary

Sets

- Analogous to mathematical notation, sets hold objects
- Similar to math, sets do not have any order as you typically (and abstractly) use them
- Sets are also wrapped around in curly brackets
- To make a set, you do `set()`

Table of Contents

Importing

- For the most part, you are going to be using your scripts modularly
- And typically, in a software engineering perspective, each script you make has a specialized purpose (which, we will go into for OOP)
- You typically don't want a large file in the event an error occurs or even if you are working with multiple people through version control (like GitHub)

Importing

- Importing is done at the top of the script as follows:

  ```
  import script
  ```

- where `script.py` is in the same directory as the current .py file you are working on

Importing: Using properties/functions from that script

- Assuming you have this at the top

```
import script
```

- You have to call properties/functions as follows:

```
script.function_1()
script.special_number
```

- Which can be a hassle to rewrite

Importing: another way

- Alternatively, you can do

```
from script import function_1, special_number
```

- Then you can call the function/property a bit more freely

```
function_1()
print(special_number)
```

Table of Contents

File handling

- For a good amount of large scaled systems, you have to deal with files
- Some files can be read, some cannot (i.e. binary files)
- But for now, we will deal with files that we can read

Opening a file in Python

- The function to open a file is: file_handle = open(`filename`, `mode`)
- The `filename` is a string for a path to a file
- The `mode` has several string values:
  - "r" = Reading
  - "w" = Writing (but first, delete everything in the file)
  - "a" = Writing, but Append to the end of the file

Closing a file

- To close a file do: `file_handle.close()`
- If you don't do this, and your code crashes, there may be a chance that you can corrupt the file

Reading a file

- `my_str = file_handle.readline()` = read one line from the file and save it into `my_str`
- `my_str = file_handle.read()` = read the whole file and save it into `my_str`
- `my_list = file_handle.readlines()` = read the whole file into a list, with each element being one line

Reading can also be done iteratively

- File handles on the mode of reading can be read in a for loop

```
for line in file_handle:
    print(line)
```

Writing into a file

- `file_handle.write()`
- Just like printing
- Except you have to add your own new line characters `"\n"`