CSC148 Ramp-up Notes

CSC148 Summer 2018

Sophia Huynh

Table of Contents

Preface

These notes are to briefly cover the topics of CSC108. They are by no means a replacement for the course itself; instead, it's to be used as a refresher for those who aren't familiar with Python but have a programming background in another language (e.g. Java), or for those who can't recall the material they learned in CSC108. As such, these notes assume at least *some* familiarity with programming.

Expectations for CSC148

For CSC148, we assume some background in programming, regardless of what language that's in. In particular, we expect you to be able to:

- Understand variables and how they work
  - E.g. Integers, strings, floats, booleans, lists (or arrays), dictionaries (or hashmaps)
  - How to assign to variables, how to use them, etc.
- Write conditional statements (if-statements)
- Write loops (both for-loops and while-loops)
- Write functions (or methods)

The remaining sections of this will provide a review of those topics, as well as a review of Python's syntax and comparisons to Java's syntax.

About Python

Python in an interpreted language, requiring no compilation to be done prior to running code. While this allows more freedom, it also means that errors resulting from incompatible types won't be noticed (i.e. you can try to add a string and an int in Python, and the error would only reveal itself during runtime). One of the freedoms that Python provides is that it's **dynamically typed**: functions can take in any type of variable, and type declarations aren't needed when declaring a variable.

Python's syntax is heavily dependent on spacing: indentation is important in determining what block a line of code belongs to, with a block of code beginning following a colon (:). In comparison, languages like Java tend to enclose blocks of code within braces ({}), with indentations being optional and only for style purposes. For example:

Python	Java

`if (x == 15):`
`    x = x + 10`
`    if (y < 15):`
`        y = y - x`
	`if (x == 15){`
	`    x = x + 10;`
	`    if (y < 15){`
	`        y = y - x;`
	`    }`
	`}`

Additionally, Python doesn't have an end-of-line character (compared to the semicolon (;) required in Java).

Variables

In Python, variables are declared by assigning a value to them without regard to their type, and all types are classes, thus having methods. Additionally, the type of a variable can change, due to the lack of type declarations in Python. For example:

Python	Java

`x = 1`
`y = True`
`x = False`
	`int x = 1;`
	`boolean y = true;`
	`// We can't do x = false`

Syntactically, declaring a variable in Python is similar to Java: Just omit the type and the semicolon.

To see what methods can be called on a variable, simply use help(`variable name`) or help(`type`).

Primitive Types

All primitive variables are immutable: the variables at that memory address cannot be modified. If a variable is given a new value, that value is created at a different memory address instead. See "Memory Model" for more details.

Integers (int)

Integers are created by simply assigning a non-decimal number to a variable (e.g. `x = 1`).

A few useful functions to know:

- abs(x): Returns the absolute value of x
- pow(x, y): Returns x to the power of y.
- x % y: Returns the remainder of dividing x by y (e.g. 5 % 2 == 1, 11 % 7 == 4).

While division (/) will always return a float (see below), integer division (denoted by //) will return a truncated integer (i.e. the result without any decimal places). For example:

```
4 // 2 == 2
4 // 3 == 1
8 // 3 == 2
```

Floats (float)

Floats are created by assigning a number with a decimal to a variable (e.g. `x = 1.0`).

A few useful functions to know:

- x.is_integer(): Returns whether x is an integer or not.
- round(x): Rounds x to the nearest integer

Any division (/) will return a float, except when integer division (//) is used. For example:

```
4 / 2 == 2.0
4 / 3 == 1.3333333333333333
8 / 3 == 2.6666666666666665
```

Booleans (bool)

Booleans are creating by assigning either True or False to a variable (e.g. `x = True`).

Boolean expressions include:

- Equality (`x == y`): True iff x and y have the same values.
- Inequality (`x != y`): True iff x and y have different values.
- Less-than, greater-than (`x < y`, `x > y`): Compares x and y; returns True iff x is strictly less-than/greater-than y.
- Less-than-or-equal-to, greater-than-or-equal-to (`x <= y`, `x >= y`): Compares x and y; returns True iff x is less-than (or equal to)/greater-than (or equal to) y.
- is (`x is y`): True iff x and y have the same memory address.
- not (`not x`): True iff x is False.
  - Inverts the truth value of x.
  - If x is not a boolean, then it returns True if it's something equivalent to False (i.e. 0, empty string, empty list, empty dictionary, None) and False otherwise.
- is not (`x is not y`): True iff x and y have different memory addresses.
- x and y: True iff both x and y are True.
- x or y: True iff at least one of x or y are True.

When trying to use a non-boolean value in a conditional statement, True is represented by any value that is not 0, not None, or not empty (e.g. an empty list [] is False).

Strings (str)

There are multiple ways to create a string:

- Encasing the string you want in a pair of single quotes (')
  ```
  x = 'cat'
  ```
  - This notation allows you to declare strings with "s in them. If you want a ' within it, you need to escape the " with a \ preceding it. For example:
    ```
    x = 'cat\'s "hat"'
    ```
- Encasing the string you want in a pair of double quotes (")
  ```
  x = "cat"
  ```
  - This notation allows you to declare strings with 's in them. If you want a " within it, you need to escape the " with a \ preceding it. For example:
    ```
    x = "cat's \"hat\""
    ```
- Encasing the string you want in a pair of triple single quotes (''')
  ```
  x = '''cat'''
  ```
  - This notation allows you to declare strings on multiple lines, also allowing 's, "s and """s to be present in the string. For example:
    ```
    x = '''cat's ... """
    "hat"'''
    ```
- Encasing the string you want in a pair of triple double quotes (""")
  ```
  x = """cat"""
  ```
  - This notation allows you to declare strings on multiple lines, also allowing 's, "s and '''s to be present in the string. For example:
    ```
    x = """cat's ... '''
    "Hat""""
    ```

Other useful ways of using a string include indexing into it, slicing it, checking if something is inside of it, and finding the length of it.

Operation	Description	Example

Indexing	Returns the character at a certain index of a string. Indexes start at 0.
	Negative indexes start counting from the end (i.e. the last character in a string is at index -1)
		`x = "cat"`
		`x[0] == 'c'`
		`x[1] == 'a'`
		`x[-1] == 't'`

Slicing	Returns the characters within a range of indexes. The first index provided is the starting point (inclusive; 0 if nothing is given), the second is the end (exclusive; the length of the string if nothing is given), and an optional third parameter represents the 'step' (how many steps to take between each letter; 1 by default).	`x = "abracadabra"`
		`x[1:3]== 'br'`
		`x[:4] == 'abra'`
		`x[3:] == 'acadabra'`
		`x[2:6:2] == 'rc'`

in	Returns whether the given string is a substring of our string.	`x = 'potato'`

	Case-sensitive.	`'p' in x == True`
		`'tat' in x == True`
		`'z' in x == False`
		`'Po' in x == False`

Length	Returns the length of a string (i.e. 1 + the last index).	`x = 'potato'`
		`len(x) == 6`

There are also many useful string methods. To see a full list of them, type in help(str) in the Python shell. A few examples are as follows:

- `str.upper()`: Returns an uppercase version of str. For example:
  - `"cAt".upper() == "CAT"`
- `str.isupper()`: Returns whether all letters in str are uppercase or not (if there are no letters in str, returns False). For example:
  - `"cAT".isupper() == False`
- `str.lower()`: Returns an lowercase version of str. For example:
  - `"CaT".lower() == "cat"`
- `str.islower()`:Returns whether all letters in str are lowercase or not (if there are no letters in str, returns False)
  - `"cAT".islower() == False`
- `str.isalpha()`: Returns whether all characters in str are letters. For example:
  - `"a123".isalpha() == False`
- `str.isalnum()`: Returns whether all characters in str are letters or numbers. For example:
  - `"a432".isalnum() == True`
- `str.isdigit()`: Returns whether all characters in str are numbers. For example:
  - `"123".upper() == True`
- `str.strip()`: Returns str with all leading and trailing whitespace (or characters in the string provided if one is given -- e.g. `str.strip("abc")` will remove all leading and trailing characters that are in 'abc'.)
- `str.split()`: Returns all of the substrings in str separated by the delimiter provided (whitespace if nothing is provided).
  - `"The cat in the hat".split() == ['The', 'cat', 'in', 'the', 'hat']`
  - `"This, is a string, with commas".split(",") == ['This', ' is a string', ' with commas']`

Lists (list)

In Python, lists are dynamic: you don't have to specify a size for them. List are denoted using square brackets (`[]`). For example:

Python	Java
`lst = [1, 2, 3]`	`int lst[] = {1, 2, 3};`

To add to the end of a list, use the list method append (e.g. `lst.append(5)`). The operations for a string (indexing, slicing, in, length) are the same for a list. Slicing a list returns a new list at a new memory address, so changes to the slice won't affect the original list.

To modify a list, you must modify the list via indexes. For example, the following modifies lst to be [0, 2, 3]:

```
lst = [1, 2, 3]
lst[0] = 4
```

While the following does not:

```
lst = [1, 2, 3]
x = lst[0]
x = 0
```

Other useful string methods are listed below. For a full list of list methods, type `help(lst)` in the Python shell.

- `list.append(x)`: Appends x to the end of the list (so list[-1] == x).
- `list.extend(x)`: Adds all elements of x, where x is a list, to the end of list.
- `list.copy()`: Returns a shallow copy of list. An alternative to this is to do list[:].
- `list.insert(index, x)`: Inserts x before index in the list.
- `list.pop(x)`: Removes and returns the element at index x in list.
- `list.sort()`: Sorts list.
    - `sorted(list)` returns a new list with sorted values, leaving list intact/unsorted.

Tuples (tuple)

Tuples are similar to lists, but they cannot be modified. They're created using round brackets (`()`). For example:

```
t = (1, 2, 3)
```

Since tuples can't be modified, none of the list methods that modify a list can be used on a tuple. To get tuple version of a list, use the `tuple()` function (e.g. `tuple(list)`). To get a list version of a tuple, use the `list()` function (e.g. `list(tuple)`).

Dictionaries (dict)

Dictionaries are similar to hashmaps. They map a key to a value, using notation similar to lists (but instead of numerical indexes, you use keys to index into the dictionary). Dictionaries are created using braces ({}). For example:

```
d = {'key': 'value,
     15: 'fifteen',
     'Yes': 3}
d['key'] == 'value'
d[15] == 'fifteen'
d['Yes'] == 3
```

To add a key into a dictionary, assign a value to a key. For example:

```
    d['new key'] == 12
```

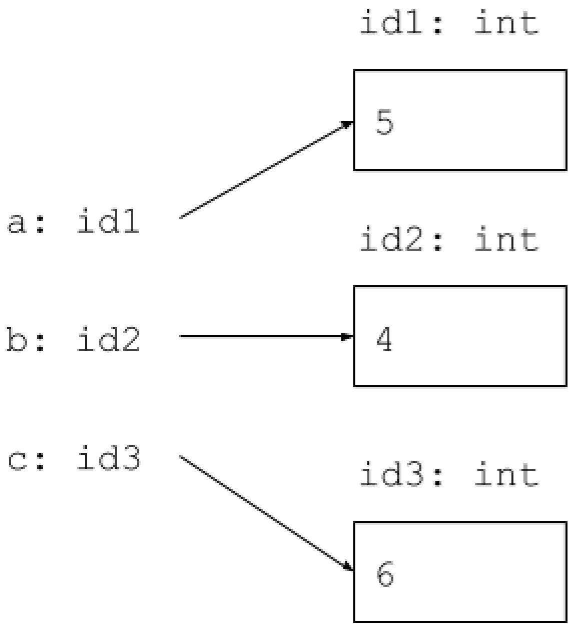
This creates the key 'new key' in the dictionary d, giving it the value 12.

If you loop over a dictionary, then you'll be looping over its keys. To get the keys of a dictionary, use the `keys` method (e.g. `dict.keys()`). For a list of all of the methods in a dictionary, use `help(dict)` in the Python shell.

Memory Model

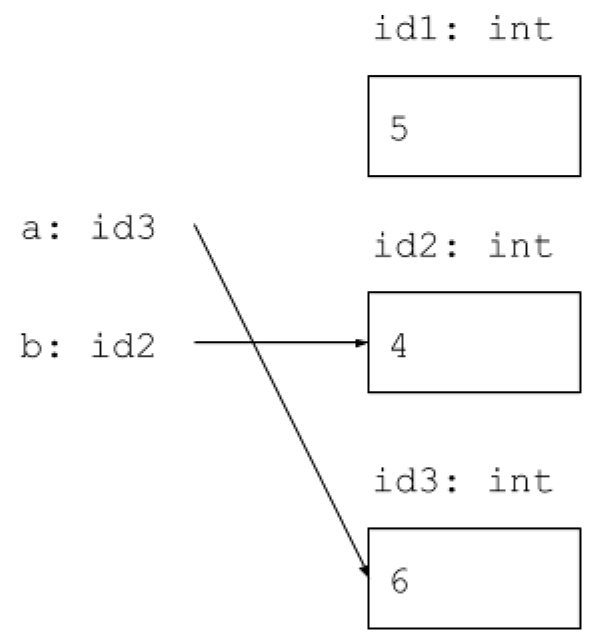
When creating variables, variable names are mapped to memory addresses containing the variable's value. For example, the code on the left would create the memory model on the right:

Code	Memory Model

`a = 5`
`b = 4`
`c = 6`
	 id1: int
	5
	a: id1
	id2: int
	b: id2
	4
	c: id3
	id3: int
	6

<table>
<tr><td>

```
a = 5
b = 4
a = 6
```

</td><td>


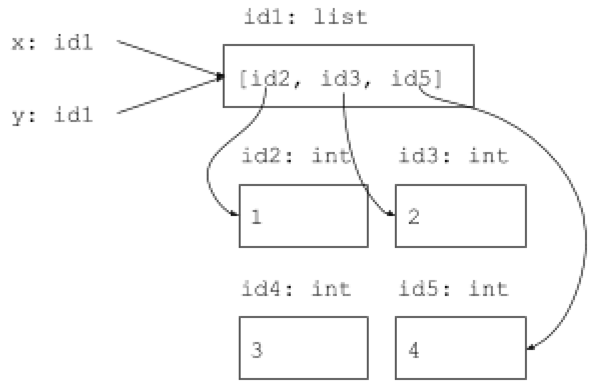
</td></tr>
</table>

Primitive values have the same id if they have the same value (i.e. the memory address of True is always the same, the memory address of 4 is always the same, and the memory address of 'cat' is always the same, regardless of how it's created). Primitive types are also **immutable**: the values at each memory address (id) cannot be changed.

For non-primitive types, such as lists, dictionaries, and classes that are defined, memory addresses are mutable, and we can refer to the same list through multiple variables ("aliases") via **aliasing**. Modifying the list through one of those variables will modify it for all variables. For example:

<table>
<tr><th>Code</th><th>Memory Model</th></tr>
<tr><td>

```
x = [1, 2, 3]
y = x
y[-1] = 4
```

</td><td>


</td></tr>
</table>

You must be extremely careful when modifying lists, dictionaries, and classes. If you don't want to modify the original version, make sure to create either a new instance of the object and update it accordingly, or create a copy.

To visualize your code, the PythonTutor may be helpful.

Conditionals (if-statements)

Conditionals in Python use the keywords **if**, **elif**, and **else**. Conditionals *must* start with an 'if', and can be followed by any number (0 ~ n) of **elifs**, and then end with 0 or 1 **else** statements. Each condition ends with a colon, and the indented block following that represents what belongs to the condition. For example:

Python	Java

`if x == 0:`
`    ...`
`elif x == 15:`
`    ...`
`else:`
`    ...`
	`if (x == 0){`
	`    ...`
	`} else if (x == 15){`
	`    ...`
	`} else {`
	`    ...`
	`}`

See booleans for more details on what can be used as conditions in an if-statement.

Loops

For-loops

In Python, for-loops iterate over sequences (e.g. lists). We can loop over strings (going through each character, starting from the first), or over lists (going through each item, starting from the first). For example:

```
for i in "cat":
    ...
```

`i` would take the value 'c' on the first iteration, 'a' on the second, and then finally 't' on the third. Similarly:

```
for i in [1, 2, 3]:
    ...
```

`i` would take the value 1 on the first iteration, 2 on the second, and then finally 3 on the third.

We can use the range() function to generate a set of numbers for us to iterate over. For example:

Python	Java

`for i in range(len(lst)):`
`    ...`
	`for (int i = 0; i < lst.length; i++){`
	`    ...`
	`}`

`range(lst)` generates a list of numbers from 0 up to (but not including) len(lst). If we want to loop over a list by indexes, that notation is how we would do it.

The `range` function has 3 forms:

- `range(x)`: Generates a list of numbers from 0 up to but not including x.
  - `range(4) == [1, 2, 3]`
- `range(x, y)`: Generates a list of numbers from x up to but not including y.
  - range(3, 5) == [3, 4]
- `range(x, y, z)`: Generates a list of numbers from x up to but not including y, taking every zth number (z = the 'step').
  - `range(1, 11, 2) == [1, 3, 5, 7, 9]`
  - `range(10, 4, -1) == [10, 9, 8, 7, 6, 5]`

The lists returned by range aren't *actual* lists, but range objects. To get a list version of a range object, just cast it into a list via `list(range)`.

While-loops

While-loops continue running as long as the condition provided evaluates to True (and, in fact, won't run at all if that condition is initially False).

Syntactically, while-loops in Python are similar to those in Java:

Python	Java

`while x:`
`    ...`
	`while (x){`
	`    ...`
	`}`

There is, however, no do-while loop in Python.

Using Functions

Function calls in Python are similar to in Java: You pass the function name followed by brackets containing the arguments to be passed in. For example:

Python	Java
`myFunction(arg1, arg2)`	`myFunction(arg1, arg2);`

Defining Functions

To define a function, we use the keyword **def**, followed by the function name, and then the parameters. No type declarations are required when defining a function.

Python	Java

`def myFunction(p1, p2):`
`    ...`
`    return x`
	`public int myFunction(p1, p2){`
	`    ...`
	`    return x;`
	`}`

By default, functions without a return statement will return None. Similar to Java, the first return statement reached will exit the function entirely.

While type declarations are not needed, annotating the types is preferred for style (see Documentation).

Importing Modules

In order to import modules (e.g. classes and functions from other files), use the **import** keyword. If you only want to import a few functions/classes, then you should also use the **from** keyword:

- `import module`: Imports everything from module (if it's not a built in module, then this would refer to the contents of module.py).
- `from module import something`: Imports something from module (or module.py).
    - You can import multiple things by using commas:
      `from module import function1, class2`
- `from module import *`: Imports everything from module. * works as a wild-card.

if __name__ == '__main__'

Within your Python file, you can include an `if __name__ == '__main__':` block. When running that file, everything within your `if __name__ == '__main__'` will be run. However, when imported by another module, nothing in the `if __name__ == '__main__'` block will run.

Code in module.py	Running module.py	module.py is imported

`def f(x):`
`    return x + 1`
	`f` refers to this function.	`f` refers to this function.

`x = 20`	`x` is now 20	`x` is now 20

`if __name__ == '__main__':`
`    x = x + 1`
	`x` is now 21.
	`f(21)` gets printed.
		This block does not run.

`print(f(x))`		

Reading and Writing Files

You must open a file before it can be used. To do so, we use the **open** keyword and assign the newly opened file to a variable. For example:

- `file = open("file.txt")`: Opens the file 'file.txt' for reading. 'file.txt' should be in the same folder/directory as the Python file from which this command was run.
    - `file = open("file.txt", "r")` does the same ('r' stands for 'read').
- `file = open("file.txt", "w")`: Opens (and creates) the file 'file.txt' for writing.

When you're done with a file, you must close it via `file.close()`. Alternatively, you can handle files within a with/as block, which closes the file for you.

```
with open('file.txt') as file:
    …
```

Reading from a File

There are multiple ways to read from a file:

- `file.readline()` will read and return the first unread line.
- `file.read()` will return everything from the first unread line to the end of the file as a single string.
- `file.readlines()` will read and return all of the lines in the file as a list of strings (with each line as its own string) starting from the first unread line.
- `for line in file`: will loop through the file. This is the same as calling:

    ```
    line = file.readline()
    while line:
        …
        line = file.readline()
    ```

Each of these methods will include the newline character ('\n') at the end of the lines. You may want to use the str.strip() method to remove these.

Writing to a File

To write to a file, you use `file.write(string)`. When writing to a file, you must manually account for the newline characters and convert whatever you write into a string.

Debugging

The method to debug a file in Python depends on the IDE you're using. In general:

- Selecting a line of code as a breakpoint is done by clicking that line number (usually to the left of the line; in WingIDE and PyCharm, breakpoints appear as red dots).

- Running the debugger will execute code in the file (usually this will refer to code in your `if __name__ == '__main__'` block).
    - The debugger usually takes the form of a green 'bug' icon, usually beside the run button (typically a green arrow).

Testing

There are multiple ways to test your code in Python (aside from manually testing in the shell). Typically testing is done through doctests or through unittests.

Doctest

Doctests run all of the docstring examples you've written for each function. To run doctest, run the lines:

```
import doctest
doctest.testmod()
```

Doctest formatting is extremely picky, and will pick up on errant spaces and different quotation marks (since it compares the result to a string). Docstring examples should be within your docstrings (see Documentation), with each 'command' that you would execute in the shell being prefaced by >>> (or … if it's a command on multiple lines/a block of code), with the expected result following it. For example:

```
def my_function(x: int) -> int:
"""
Return double the value of x.

>>> my_function(20)
40
>>> my_function(1 + 10)
  22
"""
```

The first call to my_function(20) will work correctly, and doctest won't complain. However, for the second call, doctest would fail it since there are errant spaces at the start.

Generally docstring examples/doctests inform others on how to expect your code to be used, as well as any edge cases that they might encounter.

Unittest

Unittests are written in a separate file, and tend to be more detailed and thorough than doctests. These are similar to JUnit Tests in Java. To create a unittest suite, we write a unittest class by starting with:

```
class UnitTestClassName(unittest.TestCase):
```

And within it, we define methods. Methods that start with `test` will be run and considered test cases.

Typically, we make calls to self.assert<something> within those methods:

```
    def test_something(self):
        expected = 20
        actual = my_function(10)
        self.assertEqual(expected, actual)
```

There are various assert statements you can use, such as `assertTrue`, `assertFalse`, `assertNotEqual`, etc.

To run your unittest suite, call `unittest.main()`. Typically, we do this at the end of the file in an `if __name__ == '__main__'` block:

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

See the Unittest documentation for more details: https://docs.python.org/3/library/unittest.html

Documentation

Comments

In Python, comments are denoted using # (as opposed to // in Java). Multi-line comments don't exist.

```
# This is a comment
x = 15 # They can also be placed in-line
```

Documenting Functions

When documenting functions, we tend to do so via type annotations and docstrings. When calling the help() function on another function, the docstring is provided to the caller.

Type Annotations

While there are multiple methods of documenting the types of parameters that a function takes, as well as its return type, the current method is to use Type Annotations. For parameters in a function, we follow its name with a : and the type, while return types are done following the parameters via an -> and the return type. For example:

```
def my_function(x: int, y: str) -> int:
```

int, str, bool, float are all built in types which don't need to be imported. However, the below types need to be imported to be used (i.e. via `from typing import <type>`):

- `List`: The type for lists. You can specify the types that you'd expect by using square brackets:
  `List[int]`
  - If you don't want to specify a type, you could use the built-in type `list`
- `Dict`: The type for dictionaries. You can specify the types of the key and value that you'd expect by using square brackets (in the order key, value):
  `Dict[int, str]`
  - If you don't want to specify a type, you could use the built-in type `dict`
- `TextIO`: The type for files that have already been opened (e.g. via a call to open(file)).
- `Any`: A type that captures everything. `object` can also be used here if you don't want to import Any.

If documenting a type from within that class itself, encase the class name with quotation marks ('). For example:

```
class Something:
    def something_else(self, other: 'Something'):
```

Docstrings

Docstrings are a description of **what** your function does (as opposed to **how** it does it). By convention, docstrings should mention the parameters of a function by name, as well as take the imperative mood (e.g. "Return x" as opposed to "Returns x"). They should also include an example of how the function should be called, and what its effect it. A few examples:

Good Docstring	Bad Docstring

`def function(x: int) -> int:`
`    """`
`    Return double the value of x.`
 	
`    >>> function(20)`
`    40`
`    """`
	`def function(x: int) -> int:`
	`    """`
	`    Returns double the value of x by`
	`    adding x to x and then returning`
	`    this sum.`
	
	`    >>> function(20)`
	`    40`
	`    """`
	
	This docstring is 'bad' because it goes into detail of *how* the code works, as opposed to just describing *what* it does.

`def function(x: int, y: str) -> str:`
`    """`
`    Return a string that contains`
`    y repeated x times.`
	`def function(x: int, y: str) -> str:`
	`    """`
	`    Return a string that repeats`
	`    the string given multiple times.`

`>>> function(3, 'cat')`
`'catcatcat'`
`"""`
	`>>> function(3, 'cat')`
	`'catcatcat'`
	`"""`
	This docstring is 'bad' because it doesn't mention the parameters by name; the reader may not understand which string is being repeated, nor how many times it'll be repeated.

Additional Resources

Though this document provides a fairly high-levelled and brief description of the material covered in CSC108, you might want to look into additional resources. A few of these are as follows:

- Ramp-up slides from a previous iteration of CSC148 (Winter 2018): http://www.teach.cs.toronto.edu/~csc148h/winter/rampupdata/csc148_rampup/rampup_w18.pdf
- Function Design Recipe from CSC108: http://www.cdf.utoronto.ca/~csc148h/fall/lectures/python-recap/common/design_recipe.html
    - This details how to approach designing and writing a function.
- How to Think Like a Computer Scientist (HTLCS): http://openbookproject.net/thinkcs/python/english3e/index.html
    - Covers topics within CSC108 and CSC148. The closest thing to a 'textbook'.
- CSC108 PCRS Videos: https://www.youtube.com/channel/UCu8NnRGTGxHe96Le0xqLrNQ/playlists
    - These are the videos that teach material for CSC108.

Additionally, looking through the content of past offerings of CSC108 and CSC148 might be of use too.

If you find yourself stuck, consult your TAs and instructor for help! They're also resources at your disposal.