Topic 3: Strings, Main, and Debugging.

Goals: By the end of this week, we will discuss…

- *string operations and accessing individual letters in a string*
- *the `main()` function*
- *debugging and good coding practises*

Acknowledgements: These class notes build on the content of my previous courses as well as the work of R. Jordan Crouser, Jeffrey S. Castrucci, and Dominique F. Thiébaut.

Recall: Strings

- A string is made up of characters. Characters can be letters, numbers, symbols, spaces, and escape sequences (eg. \n is the newline character, not the new line characters).
- In Python, a string is declared using quotation marks
- `str`: strings

  Words and letters: For example: 'dog', 'a', 'num', "Soddard G2"

Any questions about comparing strings? For example:

```
>>> 'a' != 'A'
True
>>> 'a' < 'A'
False
```

Operations

Concatenation: join two strings together with '+'

```
>>> "Room:" + " " + "Stoddard G2"
```

Repetition (i.e. self-concatenation): use *, e.g.

```
>>> 3 * "hi"
```

We combined these concepts in our table last week:

```
>>> print('+'+'-'*67+'+')
+-------------------------------------------------------------------+
```

Multi-line Strings

Problem: a string that looks ugly when you try to type it all on one line, e.g.

This doesn't work...syntax error.

```
desc = "This course is an introduction to computer
        science and computer programming. The programming
        language Python (Version 3) is used to introduce
        basic programming skills and techniques."
```

Three different approaches depending on preferred output:

```
desc1 = "This course is an introduction to co\
        mputer science and computer programming. The\
        programming language Python (Version 3) is u\
        sed to introduce basic programming skills an\
        d techniques."
print(desc1)

desc2 = """This course is an introduction to
        computer science and computer programming.
        The programming language Python (Version 3)
        is used to introduce basic programming
        skills and techniques."""

print(desc2)

desc3 = "This course is an introduction to computer " \
        "science and computer programming. The programming " \
        "language Python (Version 3) is used to introduce " \
        "basic programming skills and techniques."

print(desc3)
```

- We can use triple quotes to make a multi-line string.
- Use `\` to denote that the string goes over the line.

Escaping Quotes (Review)

Problem: you have a statement that contains both an apostrophe and double quotes, e.g.

```
“I can’t!” he said
```

- If we try to wrap it in single quotes, Python thinks the apostrophe in should end the string:

```
s = '“I can’t!” he said’
```

- If we try to wrap it in double quotes, Python thinks the double quote at the beginning of the sentence should end the string

```
s = ““I can’t!” he said”
```

Solution: protect (“escape”) special characters using a backslash, e.g.

```
s = '“I can\’t!” he said’
```

or

```
s = “\“I can’t!\” he said”
```

Accessing Individual Letters

One way to think about a **string** is as a **list** of letters:

```
          name = "Smith College"
     ≈ ['S','m','i','t','h',' ','C','o','l','l','e','g','e']
         0   1   2   3   4   5   6   7   8   9   10  11  12
```

- An index is a position within the string. Positive indices count from the left-hand side with the first character at index 0, the second at index 1, and so on. Negative indices count from the right-hand side with the last character at index -1, the second last at index -2, and so on.
- *Note: The first character of a string is at index 0.*
- We can access specific locations within strings with "bracket notation".
- Examples:

```
>>> name = "Smith College"
>>> name[2]
'i'
>>> name[0]
'S'
>>> name[-1]
'e'
>>> name[-4]
'l'
```

Slicing (getting a substring)

We can extract more than one character using slicing. A slice is a substring from the start index up to but not including the end index. For example:

```
>>> name = "Smith College"
>>> name[1:5]       #up to but not including 5
'mith'
>>> name[:5]
'Smith'
>>> name[2:]
'ith College'
>>> name[-2:]
'ge'
>>> name[::3]
'StCle'
>>> name[::-2]
'eelChiS'
```

Note: The slicing and indexing operations do not modify the string that they act on, so the string that `name` refers to is unchanged by the operations above. In fact, we cannot change a string. For example, this will produce an error:

```
>>> name[5] = 'a'
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    name[5] = 'a'
TypeError: 'str' object does not support item assignment
```

Strings are immutable: that means they cannot be changed... but you can still reassign them.

```
>>> name = name[:5] + "ies at c" + name[7:]
>>> print(name)
Smithies at college
```

Exercise: Batman

Given this string:

```
s = "BATMAN AND ROBIN"
```

Write a short program that uses slicing to produce:

```
BATMAN
ROBIN
BATBIN
ROB ATM
```

Strings as Objects (primer for Week 8)

“object-oriented”

- A method is a function that is applied to a particular object.

Useful methods for working with strings:

- `s.lower()`: convert the string s to lowercase
- `s.upper()`: convert the string s to UPPERCASE
- `s.strip()`: remove whitespace from the start / end of s
- `s.title()`: converts the first character in each word to UPPERCASE and remaining characters to lowercase
- `s.replace('old', 'new')`: replace all occurrences of 'old' in s by 'new'
- `s.split(c)`: slice s into pieces using c as a delimiter
- `s.join(list)`: opposite of split(), join the elements in the list together using s as the delimiter, e.g. `'-'.join(['a', 'b', 'c'])` # a-b-c
- `s.find(s1)`: returns the first index of s1, or -1 if no such index exists.
- `s.rfind(s1)`: returns the last index of s1, or -1 if no such index exists.

Reminder: Strings in python are immutable (along with ints, floats, bools, and a few other built-in types). This means that when we call a method on them, the original isn’t modified.

For example, a lowercase version of the str that white_rabbit refers to is returned when the method lower is called:

```
>>> white_rabbit = "I'm late! I'm late! For a very important date!"
>>> white_rabbit.lower()
>>> "i'm late! i'm late! for a very important date!"
>>> white_rabbit
>>> "I'm late! I'm late! For a very important date!"
```

The method `rfind()` returns the first index where the substring str is found, or -1 if no such index exists.
The method `rfind()` returns the last index where the substring str is found, or -1 if no such index exists.

```
>>> str1 = "How much wood would a woodchuck chuck if a woodchuck
could chuck wood?"
>>> str2 = "wood"
>>> str1.find(str2)
9
>>> str1.rfind(str2)
65
```

We could also replace the word, wood with something else using the method replace().
The method replace() returns a copy of the string in which the occurrences of old have been replaced with new, optionally restricting the number of replacements to max.

```
>>> str1 = "How much wood would a woodchuck chuck if a woodchuck
could chuck wood?"
>>> str1.replace("wood", "chocolate")
'How much chocolate would a chocolatechuck chuck if a chocolatechuck
could chuck chocolate?'
```

The `main()` Function

So far, we've been writing code in files as if we were writing it on the console:

```
s = "BATMAN AND ROBIN"
print(s[:6])
print(s[-5:])
print(s[:3]+s[-3:])
print(s[11:14],s[1:4])
```

When we do this, the Python interpreter executes everything from the top down.

...

It is better practice to write the code you want to execute inside a main() function, e.g.

```
def main():
    s = "BATMAN AND ROBIN"
    print(s[:6])
    print(s[-5:])
    print(s[:3]+s[-3:])
    print(s[11:14],s[1:4])

main()
```

This lets the interpreter ”read ahead” and then execute

- The interpreter reads from the top down, which means that it reads the definition first.
- Then it reads each line inside the definition, but these don't get executed yet.
- At this stage, we've given python a "recipe" for what we want it to do when we call main().
- If we just define main but don't call it, nothing happens.
- So, the real work happens only when we actually call the main() function
- When we do, python goes to the main() box and follows the instructions it finds there

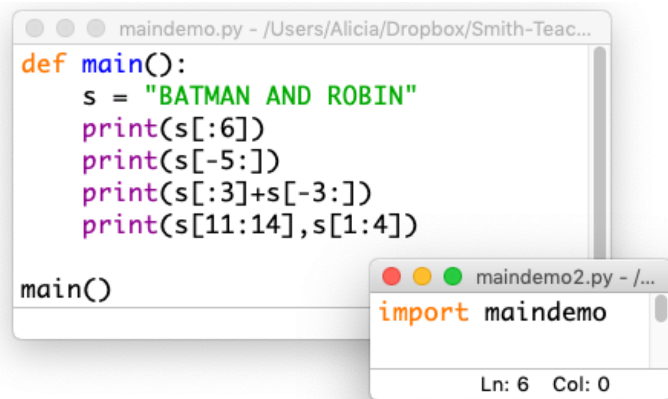
Discussion: Why bother?

- super complicated, might mess up
- code reuse -> functions()
- enable/disable chunk -> abstraction, high cohesion and low coupling
- passing parameters -> sharing information between function & readability

---

Import main()

What happens if someday we want to use the code in this file as part of another program?



What we need: a way to tell python to behave one way when we run it as a "stand-alone" program, and a different way when we import it...Ideas?

main() in Python

```
def main():
    s = "BATMAN AND ROBIN"
    print(s[:6])
    print(s[-5:])
    print(s[:3]+s[-3:])
    print(s[11:14],s[1:4])

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

- We can use an if statement to tell python to call the main() function only if the program is being run directly.
- This is a little bit confusing: we named the function we created to hold our program was called main().
- In our if statement, we're asking whether some variable called __name__ is equal to the string "__main__".
    (not to mention I don't recall initializing anything called __name__...)

To the documentation!
https://docs.python.org/3/library/__main__.html

Python » English 3.7.2 Documentation » The Python Standard Library » Python Runtime Services » previous | next | modules | index

Quick search Go |

Previous topic
builtins — Built-in objects

Next topic
warnings — Warning control

This Page
Report a Bug
Show Source

__main__ — Top-level script environment

'__main__' is the name of the scope in which top-level code executes. A module's __name__ is set equal to '__main__' when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own __name__, which allows a common idiom for conditionally executing code in a module when it is run as a script or with python -m but not when it is imported:

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

For a package, the same effect can be achieved by including a __main__.py module, the contents of which will be executed when the module is run with -m.

Python » English 3.7.2 Documentation » The Python Standard Library » Python Runtime Services » previous | next | modules | index

Quick search Go |

© Copyright 2001-2019, Python Software Foundation.
The Python Software Foundation is a non-profit corporation. Please donate.
Last updated on Feb 12, 2019. Found a bug?
Created using Sphinx 1.8.1.

Exercise:

- Write a program that contains a main() function, which contains instructions for printing out the phrase: "Only in the darkness can you see the stars."
- Use an if statement combined with checking the value of the __name__ variable to call main() only when the program is run directly.
- Add an else statement so that whenever the program ("module") is imported, it prints out the phrase: "Where are the star?"

```
def main():


if __name__ == "__main__":


else:
```

Summary

- Programs ("modules") that are well-organized are **easier to read**, more **versatile**, and potentially more **efficient.**
- The first step we'll take toward organizing our code is to include a main() function, which includes the instructions we want our program to run.
- To make it easier to import code we write now into later modules, we will follow the convention of including:

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

at the end of each module

Helpful Tip

Create a starter.py template, to make sure you don't forget any part of the main.

```
# -------------------------------------------------------
#        Name:
#     Section: L01, L02, L03, or L04
#    Filename: hmwkX.pdf
#       Peers:
#  References:
# -------------------------------------------------------


def main():
    # This is where my code will go

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