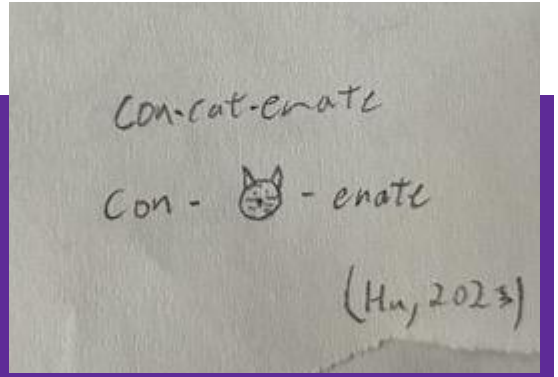
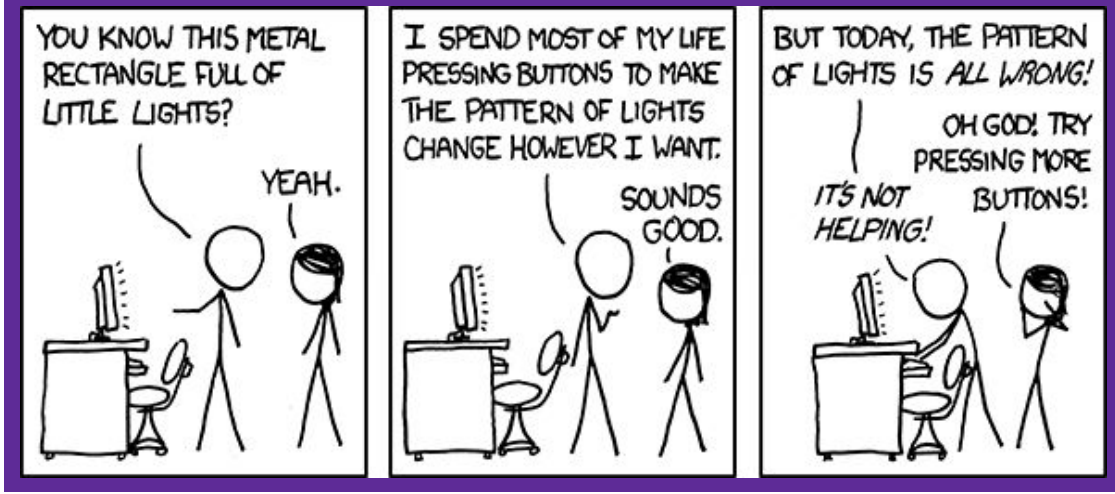
CS 1: Intro to CS

Modules and Intro to Lists





Today's Learning Objectives

Wrapping up docstrings: How and why

Modules and the `__main__` module

Lists: Our first data structure!

- Creating lists
- 0-based indexing
- Functions with lists

(Starter code and extra practice can be found on the course website)

Exit Ticket Questions

What are three takeaways from today?

What questions do you currently have?

Student question of the day: What have you found most interesting so far in this class?

Scope is Important!

So far, our variables have been defined top-down - later assignments will **shadow** earlier ones.

Functions introduce their own **local** scope - **variables inside functions only exist in during the lifetime of a function call**.

Parameters vs. Arguments

Formal parameters are simply names for the argument values passed in a function call. The **position of arguments** will determine what formal parameter name they are assigned.

They have no relationship to other variable names in the program and will override other variables if there is a naming conflict.

```
1  def f(x, y):
2      z = 2 * y # here, z is a local variable!
3      return z + x
4
5  a = 2
6  b = 20
7  ans1 = f(a, b)
8  ans2 = f(b, a) # b and a are mapped to x and y in f, respectively
```

Practice: Variables and Scoping

What is the result of executing the following program? (**`scope_mystery.py`**)

```
1  x = 1
2  y = 2
3  z = 3
4
5  def square(x):
6      return x * x
7
8  def mystery(x, y, z):
9      print('x: ', x, 'y: ', y, 'z: ', z)
10
11 mystery(x, y, z)
12 mystery(x + y, x, square(y))
```

What are three takeaways from today?

What questions do you currently have?

Student question of the day: What have you found most interesting so far in this class?

Practice: Variables and Scoping

What is the result of executing the following program?

```
1  x = 1
2  y = 2
3  z = 3
4
5  def square(x):
6      return x * x
7
8  def mystery(x, y, z):
9      print('x: ', x, 'y: ', y, 'z: ', z)
10
11 mystery(x, y, z)
12 mystery(x + y, x, square(y))
```

```
Output:
x:  1 y:  2 z:  3
x:  3 y:  1 z:  4
```

Strings are Objects

Python is what's called an **object-oriented** language (we'll learn more about what this means in upcoming lectures)

Most data types are represented as "objects"

An "object" is some **data** with associated **methods** (similar to functions) that work on that data

Python strings are an example of an object

Functions vs. Methods

Functions that are associated with an object are called **methods**

Methods are called on an object using what's called "dot-syntax"

```
>>> 'hello world'.upper()
'HELLO WORLD'
>>>
```

Compare this with the **function `print`**, which takes values (including objects) as arguments:

```
>>> print('hello world')
hello world
>>>
```

Check Your Understanding

Assume the string variable s is defined. Which of the following are function calls?

```
len(s)
s.upper()
s.lower()
help(str)
print(s)
input(s)
"hello {}".format(s)
```

Check Your Understanding

Assume the string variable s is defined. Which of the following are function calls?

```
len(s)
s.upper() # method
s.lower() # method
help(str)
print(s)
input(s)
"hello {}".format(s) # method
```

Comments (from Week 1)

Comments are lines in the source code that are notes to the reader(s), while Python just ignores them

Comments start with # and continue to the end of the line

```
# U.S. dollars per hour
salary = 18.5 # everything after the comment symbol is ignored
```

We'll learn other ways to document your code properly ~~next week~~ **today**!

Docstrings

Comments are commonly used to describe what a function does:

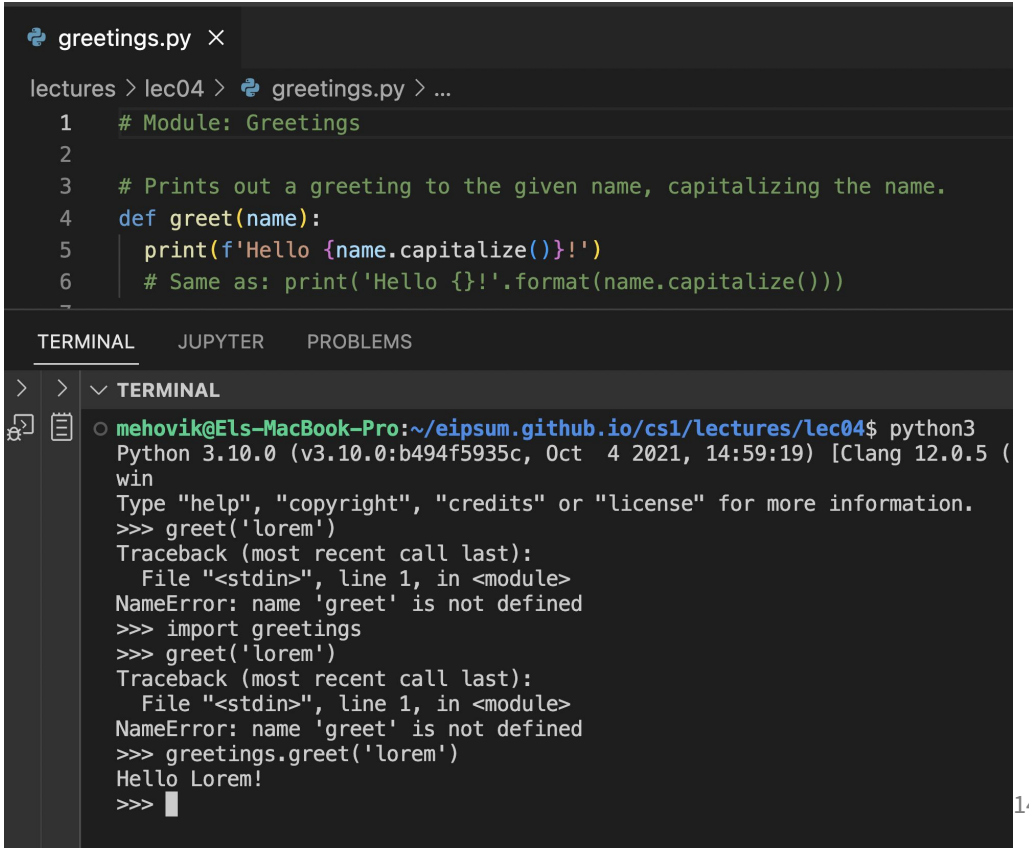
```
# Prints out a greeting to the given name, capitalizing the name.
def greet(name):
  print(f'Hello {name.capitalize()}!')
  # Same as: print('Hello {}!'.format(name.capitalize()))
```

However, Python's **`help()`** function can't use them unless we wrote them in a special way (**docstrings!**)

Aside: Loading Functions in the >>> Interpreter

Remember that running python3 to open a new interpreter *does not* mean that any of your functions written in a file are loaded

To load your functions and test them in the console, you'll need to import them (we'll learn more about **`import`** shortly):



Docstrings

A **docstring** is a regular Python string that is the first thing in any of:

- A function body
- A module
- A class (later in course)

Just like the comments we've seen so far, the docstring doesn't do anything when the program is executed

- But Python stores it as part of the function/module/class

See CS 1 Code Quality Guide for more notes/expectations on docstring format/contents.

Upgrading to Docstrings

```
# Prints out a greeting to the given name, capitalizing the name.
def greet(name):
  print(f'Hello {name.capitalize()}!')
```

```
def greet(name):
  """
  Prints out a greeting to the given `name` (str), capitalizing the name.
  """
  print(f'Hello {name.capitalize()}!')
```

Modules (Reading 6)

A chunk of code that:

- Exists in its own file
- Is intended to be used by Python code outside itself using "imports"
- Functions in imported modules become available to code that imports them

Often called “libraries” with the analogy that you can “check-out” functions/values that you need in your programs

What's in a Module?

Can contain any Python code

Most often, modules contain:

- Functions (e.g. **`math.sqrt()`**)
- Values (constants, e.g. **`math.pi`**)
- Classes (we don't know what these are yet, but we will soon!)

For now, we'll mainly be interested in modules that contain functions

Some Useful Modules

`math`: standard math functions and values

`cmath`: complex number math

`string`: string functions and values

`random`: random numbers

`sys`: system-specific functions and values

`time`: time-specific functions and values

`os`: operating system interface

`email`: email parsing

`HTMLParser`: web page processing

A list of some other interesting/fun modules can be found here and a full list is here.

Using Modules: import

There are various ways to import modules

```
>>> import math # imports a module containing useful mathematical functions
>>> math.sqrt(2.0)
1.4142135623730951
>>>
```

Note: The dot-syntax is used on Modules since they are treated as Python objects (where functions are "methods" belonging to the module)

Using Modules: import

(Demo with VSCode debugger; see **`module_demo.py`**)

More ways to import

Writing **`math.sqrt`** can be pretty verbose - is there a shorter way?

Instead, we can do:

```
>>> from math import sqrt
>>> sqrt(2.0)
1.4142135623730951
>>>
```

More concise, but not always a good thing!

Common Pitfalls

1. Using a module without importing it

```
>>> math.sqrt(2.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>>
```

Common Pitfalls

2. Not referencing the module when using an imported function

```
>>> import math
>>> sqrt(2.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>>
```

Multiple imports

Can import more than one module at a time

```
>>> import math, string, time
```

Now, can use any function in the math, string, or time modules (e.g. **`math.sqrt`**, **`string.capitalize`**, **`time.localtime`**)

Can also import more than one name from a particular module at a time

```
>>> from math import sin, cos, tan
```

Now can use **`sin`**, **`cos`**, and **`tan`** without **qualifying** them

Multiple imports

Can import *all* names from a module

```
>>> from math import *
>>> sqrt(2.0)
1.4142135623730951
>>> factorial(5)
120
>>>
```

The * means "every name in the module"

Now can use *any* function in the module without qualifying the name

This isn't always a good thing, and you can read more about why in Reading 6

`import as`

A convenient variation of the **`import`** statement

```
>>> import math as m
>>> m.sqrt(2.0)
1.4142135623730951
>>>
```

The **`as m`** means you can qualify the name with just the prefix **`m`** (or whatever name you choose) instead of the full module name (e.g. **`math`**).

Module Docstrings

We can also have docstrings for an entire module:

```
"""Module: greeetings
Functions to print out greetings."""

def greet(name):
  """Prints out a greeting to the given name, capitalizing the name"""
  print("Hello {}!".format(name.capitalize()))

def farewell(name):
  """Prints out a farewell to the given name, capitalizing the name"""
  print("Goodbye {}!".format(name.upper()))
```

Module Docstrings

```
>>> help(greetings)
```

```
Help on module greetings:

NAME
    greetings

DESCRIPTION
    Module: greeetings
    Functions to print out greetings.

FUNCTIONS
    farewell(name)
        Prints out a farewell to the given name, capitalizing the name

    greet(name)
        Prints out a greeting to the given name, capitalizing the name

FILE
    /Users/mehovik/work/cs1/lecture3/greetings.py
```

Why use docstrings?

We expect you to use docstrings for all your functions and modules

Docstrings are good documentation for:

- You now
- You in the future
- Anyone else that wants to use your modules/functions

Python can also easily turn docstrings into web pages for easy browsing

Docstrings for Functions vs. Modules

For **functions**, a docstring should describe:

1. What the function does
2. What the function **arguments** represent
3. What the function **return value** represents

For **modules**, a docstring should describe:

1. The purpose of the module
2. General description of the kinds of functions in the module (but *not* a detailed description of each function)
3. Any other relevant information

Module docstrings

We can also have docstrings for an entire module:

```
"""Module: greeetings
Functions to print out greetings."""

def greet(name):
  """Prints out a greeting to the given name, capitalizing the name"""
  print("Hello {}!".format(name.capitalize()))

def farewell(name):
  """Prints out a farewell to the given name, capitalizing the name"""
  print("Goodbye {}!".format(name.upper()))
```

Module docstrings

```
>>> help(greetings)
```

```
Help on module greetings:

NAME
    greetings

DESCRIPTION
    Module: greeetings
    Functions to print out greetings.

FUNCTIONS
    farewell(name)
        Prints out a farewell to the given name, capitalizing the name

    greet(name)
        Prints out a greeting to the given name, capitalizing the name

FILE
    /Users/mehovik/work/cs1/lecture3/greetings.py
```

The `__main__` module

Suppose we define a function with a docstring in the Python shell and try to get its documentation:

```
>>> def double(x):
...     '''Returns the argument x doubled.'''
...     return 2 * x
...
>>> help(double)
```

```
Help on function double in module __main__:

double(x)
    Returns the argument x doubled.
```

The `__main__` module

`__main__` is the name Python gives to either:

- The interactive interpreter (as in the previous slide)
- The module which was directly invoked by Python

All other modules are referred to by their own names (e.g. the **`greetings`** module)

You can get the current module with:

```
print(__name__)
```

Try it out in 1) the interpreter, 2) within a .py program that is ran with python **`filename.py`**, and 3) within a .py program that is imported with **`import filename`**

`__builtins__`

Python contains quite a few built-in functions we've already seen

- `abs`, `max`, `min`, etc.

These functions live in a special module called **`__builtins__`**

To get documentation on all of them:

```
>>> help(__builtins__)
```

A New Sequence: List

Recall that a string is a **sequence** of characters

```
my_str = "abcde"
```

A list is a sequence of *any* kind of value

```
my_lst = ["a", "b", "c", 1, 2, 3]
```

… even emojis!

```
>>> import emoji
>>> duck_emoji = emoji.emojize(':baby_chick:')
>>> duck_list = [duck_emoji] * 10
>>> duck_list
['🐤', '🐤', '🐤', '🐤', '🐤', '🐤', '🐤', '🐤', '🐤', '🐤']
>>>
```

Code Demo

There's quite a lot we can do with lists, so let's jump into some code to explore…

See `.py` files under today's lecture for the code and some practice exercises!

- **`lec05_lists.py`** (list demo code)
- **`extra_list_practice.py`** (3 practice exercises)
- **`duck_loop.py`** (complete duck loop program with emoji module, previewing loops for Monday)

We have also provided a **`Lec05Lists.java`** analog to **`lec05_lists.py`** (you aren't expected to know the Java code, but students have shared it's been helpful to compare the two languages!)