CSC148 Ramp-up
Winter 2017

Jaisie Sin

Credit to: Michael Kimmins, Orion Buske, Velian Pandeliev, Jonathan Taylor, Noah Lockwood, software-carpentry.org, and Julianna Paprakis

Overview

- In the next 6 hours, we’ll cover the background required for CSC148.
- This session is for students with programming experience – we will not cover basic programming concepts, it will mostly be about Python-specific concepts.
- Please ask questions!

Outline

- Week 1 Administration Recap
- Quick intro to basics
- Blueprint of a Python file
- Learn to speak Python
- Mutability & Aliasing
- Debugging
- Files - Reading & Writing
- Unit Testing

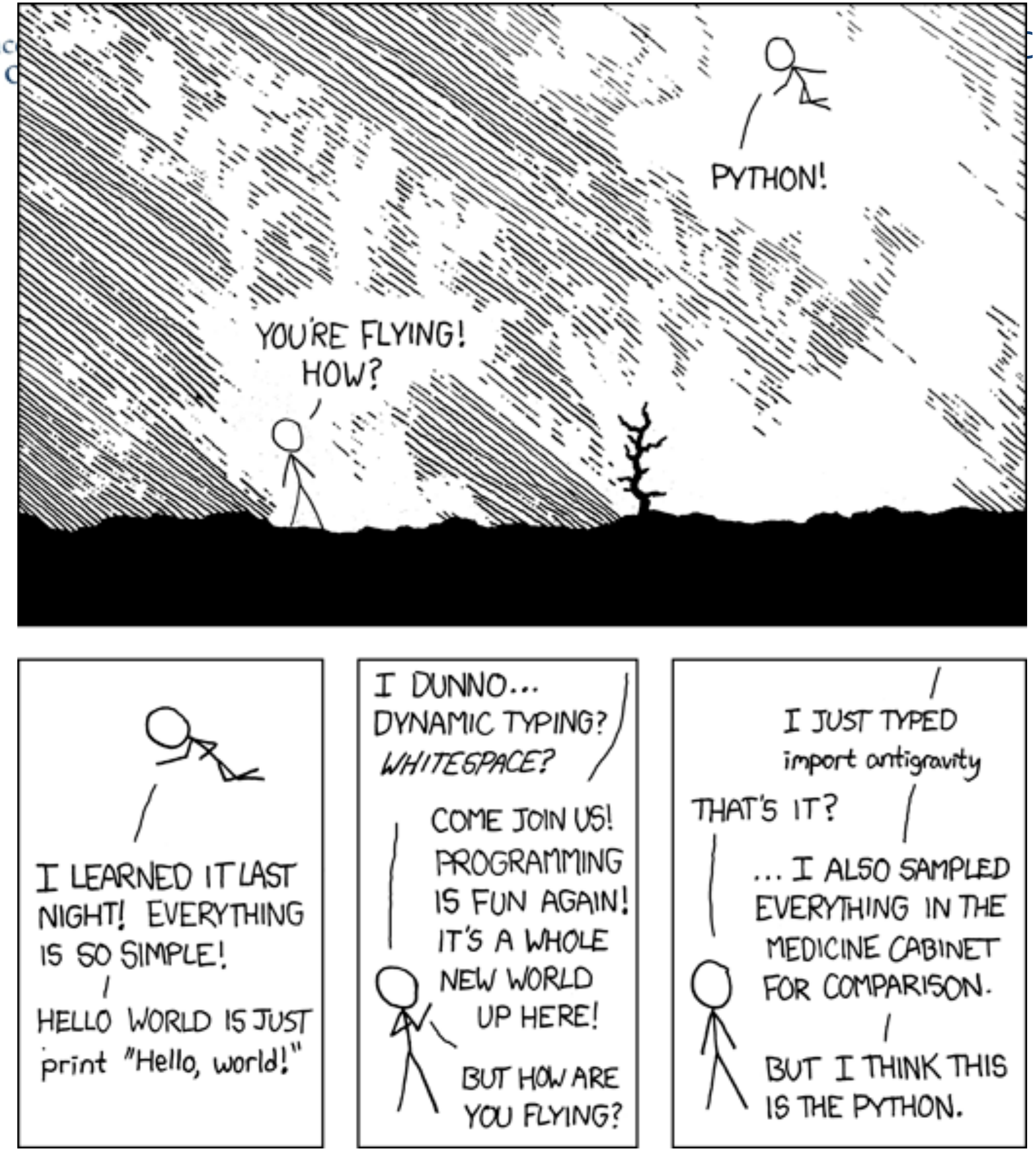
Administration

- About Me
  - What to call me: Jaisie (pronounc. ”JC”), hey you
  - Research interests: connections between computers, healthcare, and humans
  - First (programming) language: Java, then Python
- About You
  - What programming languages?

Administration

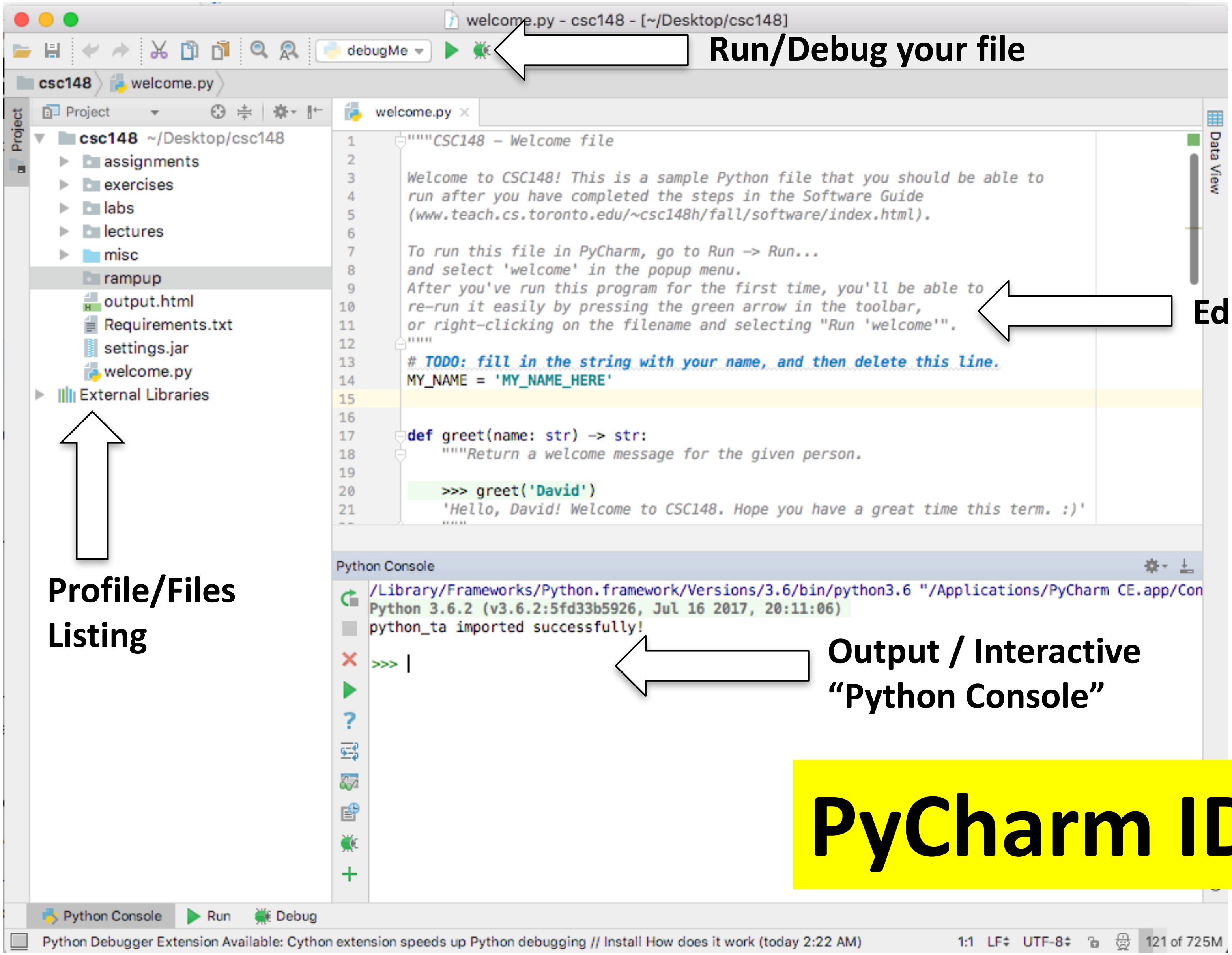
- We're using the teaching lab environment to run your programs
  - Info for new students: http://www.teach.cs.toronto.edu/resources/intro_for_new_students.html
  - Python version 3.6 - PLEASE USE THIS!!!
  - https://www.python.org/downloads
- Using the PyCharm IDE
  - More on this coming soon!
  - Installation instructions: http://www.teach.cs.toronto.edu/~csc148h/fall/software/index.html

Intro & Basics


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

How to run Python

- Programs are stored in .py files
- Edit and run your program using an IDE (Integrated Dev. Environment) like PyCharm
- You can also use the Python Console in PyCharm to run an interactive “shell”
  - Result is automatically shown (don’t need to “print” it like you would in a script)



```
"""CSC148 — Welcome file

Welcome to CSC148! This is a sample Python file that you should be able to
run after you have completed the steps in the Software Guide
(www.teach.cs.toronto.edu/~csc148h/fall/software/index.html).

To run this file in PyCharm, go to Run -> Run...
and select 'welcome' in the popup menu.
After you've run this program for the first time, you'll be able to
re-run it easily by pressing the green arrow in the toolbar,
or right-clicking on the filename and selecting "Run 'welcome'".
"""
# TODO: fill in the string with your name, and then delete this line.
MY_NAME = 'MY_NAME_HERE'


def greet(name: str) -> str:
    """Return a welcome message for the given person.

    >>> greet('David')
    'Hello, David! Welcome to CSC148. Hope you have a great time this term. :)'
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 "/Applications/PyCharm CE.app/Con
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
python_ta imported successfully!

>>> |
```

Python Debugger Extension Available: Cython extension speeds up Python debugging // Install How does it work (today 2:22 AM)   1:1 LF÷ UTF-8÷   121 of 725M

Run/Debug your file

Editor

Profile/Files Listing

Output / Interactive “Python Console”

PyCharm IDE

IDE: Interactive Development Environment

Using Python Console

- Import all the functions from a file `filename`

```
>>> import filename
```

- Import function with name `function_name` from file with name `filename`

```
>>> from filename import function_name
```

  – Don’t add the .py part of the filename

- Press up key to view past commands

The blueprint of a Python file:

```
from random import randint
from math import cos
```

import names from other modules

```
def my_function(arg):
    ...
    return answer

class MyClass:
    ...
```

define functions and classes

```
if __name__ == '__main__':
    my_variable = 21 * 2
    ...
```

your main block goes down here!

Modules (why reinvent the wheel?)

Python has a spectacular assortment of **modules** that you can use (you have to import their **names** first, though)

```
>>> from random import randint  # now we can use it!
>>> randint(1, 6)  # roll a die
4  # http://xkcd.com/221/
>>> import math
>>> math.sqrt(2) # note you have to say math.sqrt
1.4142135623730951
>>> from math import cos
>>> cos(0) # now we don’t have to use math.cos
1.0
>>> import datetime
>>> dir(datetime)
```

Demo Time!

- Basic Python Operations
- Running vs. importing a file
- Code in the main block only executes when running

Let's speak some Python

- **Interpreted** (no compilation necessary)
- **Whitespace** matters (4 spaces/1 tab for indentation)
- No end-of-line character (no semicolons!)
- No extra code needed to start (no "public static ...")
- **Dynamically typed** (a function can take multiple different types, have different behaviors)
- **Strongly typed** (all values have a type)
- `# Comments start with a '#' character.`

Where to find Documentation

- Official Python documentation:

```
http://docs.python.org/py3k/library/
```

- The `help` function provides usage information:

```
>>> help(print)
```

- The `dir` function shows names within a given type, module, or object:

```
>>> dir(str)
```

More resources!

Last term's 108 and 148 course websites:

- http://www.teach.cs.utoronto.ca/~csc108h/summer
- http://www.teach.cs.utoronto.ca/~csc148h/winter

(Easy to google these)

Online:

- https://www.tutorialspoint.com/python/
- http://greenteapress.com/wp/think-python/
- https://python.swaroopch.com/
- http://www.openbookproject.net/thinkcs/python/english3e/

Google!

- http://lmgtfy.com/?q=python+add+to+list

Learn to speak good Python

Python's style guide:

- http://www.python.org/dev/peps/pep-0008/
  - pothole_case (instead of CamelCase)

PyTA:

- www.cs.toronto.edu/~david/pyta/quick_start.html
- PyTA is derived from Pylint:
  - https://www.pylint.org

Python-Specific Syntax

- Numbers: `int` `float`
- Booleans: `True` `False`
- Operators: `or` `and` `not`
- Null: `None`
- Type Conversions:
  - `str(5.33)` gives `'5.33'`
  - `int('5')` gives `5`
    - Note: `int('5.33')` gives a ValueError!
  - `float('5.33')` gives `5.33`

Python Strings

- A string is an **immutable sequence** of **characters**
- Single quotes (') OR double quotes (") - both work
- No char/character type

String Operation	Example

Indexing	`>>> phrase = 'big orange cat'`
	`>>> phrase[2]`
	`'g'`

Slicing	`>>> phrase[3:8]`
	`' oran'`
	`>>> phrase[8:3:-1]`
	`'naro '`

in	`>>> 'g' in phrase`
	`True`
	`>>> 'z' not in phrase`
	`True`

len	`>>> len(phrase)`
	`14`

Python String Methods

- Lots of useful str methods too

```
>>> str1 = 'Hello world!'
>>> str1.islower()
False
>>> str1.lower()
'hello world!'
>>> str1.isalpha()
False
>>> str1.split() # gives words (tokens) in str1
['Hello', 'world!']
```

Sequences: [Lists]

- A list is a **mutable sequence** of **any object**

```
>>> random_stuff = [42, 3.14, 'carpe diem']
>>> random_stuff[0] = 'Replaced!'
['Replaced!', 3.14, 'carpe diem']
```

- Operations: very similar to strings:

```
>>> random_stuff[0]   # indexing returns the element
'Replaced!'
>>> random_stuff[2:]  # slicing always returns a sub-list
['carpe diem']        # as a new list
>>> random_stuff[:]   # this returns whole list as a new list
['Replaced!', 3.14, 'carpe diem']
>>> 3.14 in random_stuff
True
```

[Lists, of, things].stuff()

- Lots of other useful functions, too

```
>>> marks = [74, 62, 54]
>>> len(marks)    # gives size of list
3
>>> marks + [1, 2]# concatenation
[74, 62, 54, 1, 2]    # new list
>>> marks.pop(1)  # remove/return val at [1]
62
>>> marks.append(100) # modifies original list
>>> marks
[74, 54, 100]
```

Sequences: (tuples)

- Tuples are like lists, but are **immutable**

```
>>> stuff = (42, 3.14, 'carpe diem')
>>> stuff[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

- Can always create a list from them:

```
>>> stuff_as_list = list(stuff)
```

{'dictionaries': 'awesome'}

- **Dictionaries** (type **`dict`**) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
  - name -> phone number
  - phone number -> name
  - student id -> grade
  - grade -> ~~student id~~ list of student ids
    - Can have more than one student with the same grade
- Keys must be **unique** and **immutable**
  - Commonly strings

{'dictionaries': 'awesome'}

```
>>> scores = {'Alice': 90, 'Bob': 76, 'Eve': 82}
>>> scores['Alice']  # get
90
>>> scores['Charlie'] = 64  # set
>>> scores.pop('Bob') # delete and return removed value
76
>>> 'Eve' in scores  # membership testing
True
>>> len(scores) # number of keys
3
>>> scores == {'Bob': 76, 'Alice': 90 , 'Eve': 82}
True
# == checks equality of contents
```

For loops!

- **For loops** repeat some code for **each** element in a sequence
  - This is a foreach loop in most languages

```
>>> colours = ['red', 'green', 'blue']
>>> for colour in colours:
...     print(colour)
...
red
green
blue
```

For loops!

- Looping over characters in string:

```
>>> colour = 'red'
>>> for c in colour:
...     print(c)
r
e
d
```

- Looping over keys in a dict:

```
>>> scores = {'Alice': 90, 'Bob': 76, 'Eve': 82}
>>> for name in scores:  # loops over the keys
...     print(f'{name}: {scores[name]}')
...
Charlie: 64
Alice: 88
Eve: 82
```

For loops!

- But wait, I actually *wanted* the index!
  - Use **range**(n) in a for loop to loop over a range of numbers

```
>>> for i in range(2):
...     print(i)
0
1
```

  - To start at a value other than 0:

```
>>> for i in range(4, 6):
...     print(i)
4
5
```

Exercise 1.1: Dictionaries and Simple Formatting

- Complete 1.1 – Dictionaries and Simple Formatting on the exercise sheet

Exercise 1.1: Solution

```
for student in students:
    print(f'{students[student]} (#{student})')
```

While loops!

- **While loops** keep repeating a block of code while a condition is `True`

```
# What does this code do?
val = 10
while val > 0:
    print('hello')
    val -= 1
```

While loops!

- **While loops** keep repeating a block of code while a condition is `True`

```
# What does this code do?
val = 10
while val > 0:
    print('hello')
    val -= 1

# prints 'hello' 10 times
```

Conditionals (if, elif, else)

- **If statements** allow you to execute code sometimes (based upon some **condition**)
- **elif** (meaning 'else if') and **else** are optional

```
if amount > balance:
    print('You have been charged a $20' +
          ' overdraft fee. Enjoy.')
    balance -= 20
elif amount == balance:
    print('You're now broke')
else:
    print('Your account has been charged')

balance -= amount  # deduct amount from account
```

Exercise 1.2: Dictionaries – Dictionaries and Loops

- Complete 1.2 – Dictionaries and Loops on the exercise sheet

Exercise 1.2: Solution

```
letters = {}

for line in lst:
    for word in line.split():
        if word not in letters:
            letters[word] = 0
        letters[word] += 1
```

```
# Alternative Solution:
letters = {}

for line in lst:
    for word in line.split():
        if word in counts:
            letters[word] += 1
        else:
            letters[word] = 1
```

Functions

- Allow you to group together a bunch of statements into a block that you can call.

```
def celsius_to_fahrenheit(degrees: float) -> float:
    return (9 / 5) * degrees + 32

temp_in_f = celsius_to_fahrenheit(100.0)
```

- **Important**: If you don't specify a return value, it will be None

Functions – Design Recipe

1. **Example** Calls: doctests – will cover more of this later

   >>> celsius_to_fahrenheit(10)

   50

2. **Header**

```
def celsius_to_fahrenheit(degrees: float) -> float:
```

Functions – Design Recipe (cont.)

3. Description: **what** the function does, **not how** it does it

```
 Convert degrees from C to F.
```

4. Body: The actual function code

```
 return (9 / 5) * degrees + 32
```

5. Test Test Test Test Test – More on this later ☺

Functions – Design Recipe (cont.)

- The part of the function in triple-quotes is the **docstring**
  - It is shown when **help()** is called on your function
- Putting it all together we get:

```
def celsius_to_fahrenheit(degrees: float) -> float:
    """Convert degrees from C to F.

    >>> celsius_to_fahrenheit(10.0)
    50.0
    """
    return (9 / 5) * degrees + 32
```

1. Example
2. Header
3. Description
4. Body
5. Test

Exercise 2.1: Functions– Simple Function Reuse

- Complete 2.1 – Simple Function Reuse on the exercise sheet

Exercise 2.1: Functions– Simple Function Reuse

```
def to_listing(first_name: str, last_name: str,
               phone_number: str) -> str:
    """Return a string in the format last_name,
    first_name: phone_number

    >>> to_listing('Jaisie', 'Sin', '416-555-5555')
    'Sin, Jaisie: 416-555-5555'
    """

    return format_name(first_name, last_name) + ': '
    + phone_number
```

Memory & Mutability

- Understanding how memory works will let you know how your code will behave
- There are key differences in the behaviour of mutable objects (e.g. lists) and immutable objects (e.g. strings, tuples)
- Items in a mutable object can be changed
- Items in an immutable object cannot be changed; a new object is created in memory

- **Read the assigned reading on The Memory Model**

Memory Model - Data

- All data have three components: id, type, and value

```
>>> num = 5
>>> id(num)
4297370816
>>> type(num)
<class 'int'>  # id: 4297370816, type: int, value: 5

>>> text = 'asdf'
>>> id(text)
4327296896
>>> type(text)
<class 'str'> # id: 4327296896, type: str, value: 'asdf'
```

Mutable vs. Immutable

Mutable (e.g. list)	Immutable (e.g. str)

`>>> lst = ['Hello']`
`>>> id(lst)`
`4348611592`
`>>> lst.append('there!')`
`>>> lst`
`['Hello', 'there!']`
`>>> id(lst)`
`4348611592 # same`
	`>>> string = 'Hello'`
	`>>> id(string)`
	`4327355872`
	`>>> string = string + ' there!'`
	`>>> string`
	`'Hello there!'`
	`>>> id(string)`
	`4346210544 # different`

• The old list object could be directly changed	• The old str object couldn’t change, so Python made a new str object

Aliasing and Mutation

- Example of Aliasing:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
```

- It becomes possible in this case to modify another variable’s value:

```
>>> z[1] = 'b'
>>> x
[1, 'b', 3]
>>> y
[1, 2, 3]
```

Aliasing and Mutation – Watch out!

- Another example of referring to (and mutating) the **same mutable** data structure:

```
>>> sorted_list = [1, 2, 3]
>>> not_a_copy = sorted_list  # not a copy
>>> not_a_copy.append(0)
>>> sorted_list
[1, 2, 3, 0]  # oops

>>> actually_a_copy = list(sorted_list)
>>> another_copy = sorted_list[:]
```

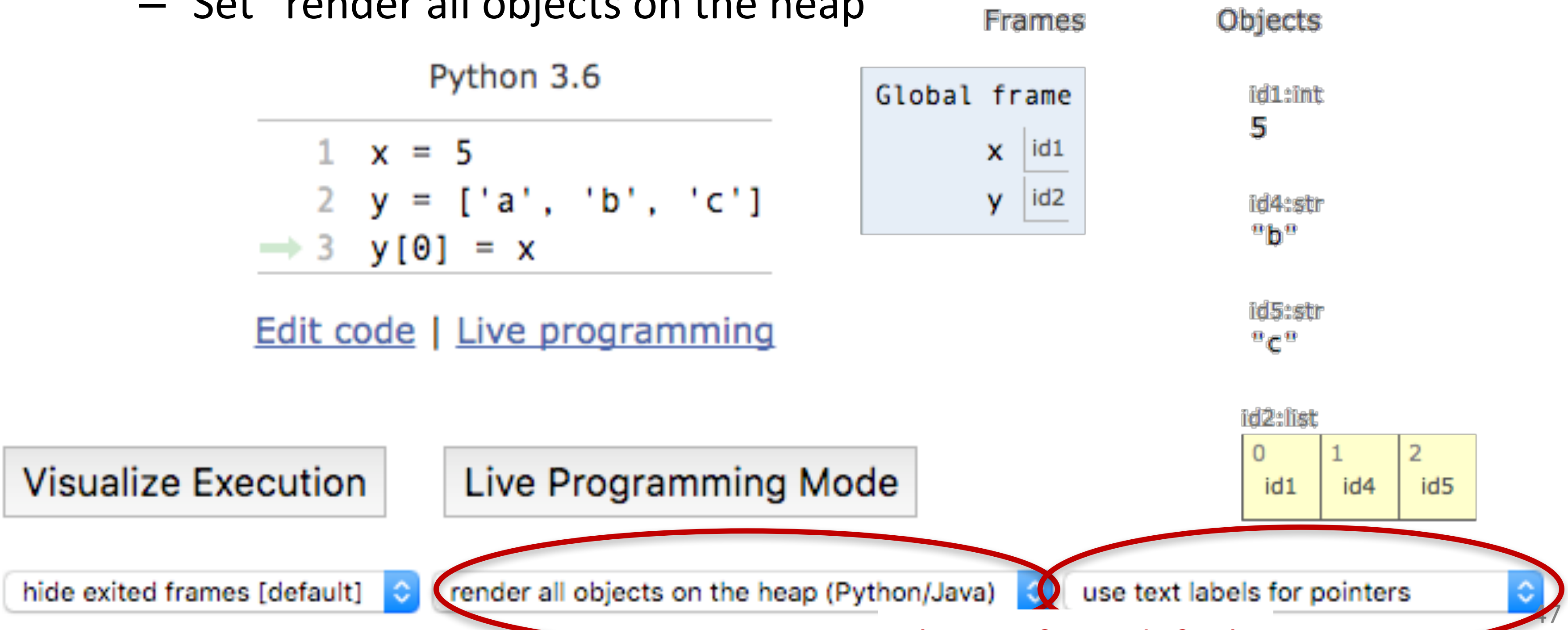
Aliasing and Mutation – Watch out!

- To prevent mutating the original data structure:

```
>>> sorted_list = [1, 2, 3]
>>> actually_a_copy = list(sorted_list)
>>> # another_copy = sorted_list[:]
>>> actually_a_copy.append(0)
>>> actually_a_copy
[1, 2, 3, 0]
>>> sorted_list
[1, 2, 3] # yay!
```

Memory Model

- You can model how your program's memory will look
- Use the Python visualizer at http://www.pythontutor.com/visualize.html
  - Set language to Python 3.6
  - Set "render all objects on the heap"



Python 3.6

```
x = 5
y = ['a', 'b', 'c']
y[0] = x
```

Frames

Global frame
x id1
y id2

Objects

id1:int
5

id4:str
"b"

id5:str
"c"

id2:list

0	1	2
id1	id4	id5

hide exited frames [default]

render all objects on the heap (Python/Java)

use text labels for pointers

Change from default

Exercise 3.1: Memory & Mutability – Variable Assignment

- Complete 3.1 – Variable Assignment on the exercise sheet

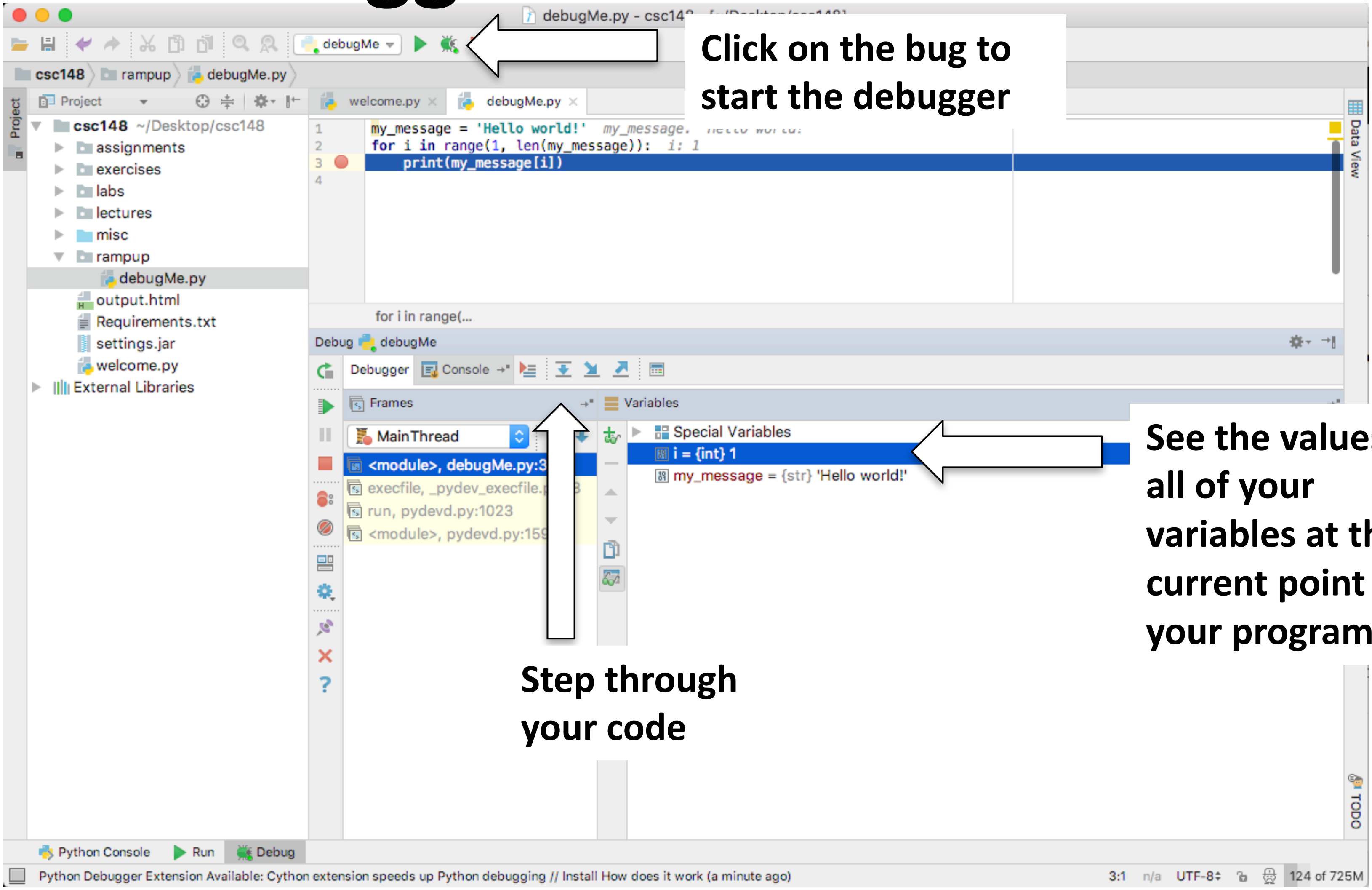
Exercise 3.1: Solution

```
a: [0, 1, 10, 4]

b: [0, 1, 10, 4]

c: 20

d: [0, 1, 10, 4]
```

Let's visualize this!!

- https://goo.gl/EKBLMW

The Debugger



```
my_message = 'Hello world!'   my_message: 'Hello world!'
for i in range(1, len(my_message)):   i: 1
    print(my_message[i])
```

for i in range(...

Frames:

```
<module>, debugMe.py:3
execfile, _pydev_execfile.py:18
run, pydevd.py:1023
<module>, pydevd.py:159
```

Variables:

```
i = {int} 1
my_message = {str} 'Hello world!'
```

Python Debugger Extension Available: Cython extension speeds up Python debugging // Install How does it work (a minute ago)

Click on the bug to start the debugger

See the values of all of your variables at the current point in your program

Step through your code

Memory Model & Equality

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
>>> id(x)
4401298824
>>> id(y)
4404546056
>>> id(z)
4401298824
```

```
>>> x == y
True
>>> x == z
True
>>> x is y
False
>>> x is z
True
```

- `==` checks for **value equality**
- `is` checks for **identity equality**

Standard input/output

- Generating output (stdout): **`print()`**
  - Can take multiple arguments (will be joined with spaces)
  - print() doesn’t return a value
- Reading keyboard input: **`input()`**

```
>>> name = input()
Jaisie  # user inputted
>>> name
'Jaisie'
>>> print('Hello ' + name)
Hello Jaisie
>>> f'Hello {name}'
'Hello Jaisie' # Why quotes here?
>>> printed_name = print(f'Hello {name}')
Hello Jaisie   # It’s printed here because print was called
>>> printed_name # What would be the next line?
```

Working with files: safely opening files

- A file must be opened before it can be used
- Use `with/as` to open something for a while, and always close it, even if something goes wrong.
- Reading Files:

```
with open('myfile.txt') as file:
    ... # do something with file (see next slide)
```

- Writing Files:

```
balance = 40
with open('output.txt', 'w') as file:
    file.write('I can write\n')
    file.write(f'Account balance {balance}\n')
```

- Access modes: 'r' for reading (default), 'w' for writing, 'a' for appending

Extra notes on reading files

- Several ways to read files:
  - With a for loop:

```
for line in f:
    ... # do something with line in file f
```

  - With file methods:
    - f.readline() - reads a single line

```
line = f.readline()
```

    - f.readlines() - reads all lines starting after last read giving a list of lines

```
lines = f.readlines()
```

    - f.read() - as a single string, starting after last line read

```
wholefile = f.read()
```

Exercise 4.1 & 4.2: Reading/Writing Files

- Complete 4.1 – Writing to a file
- Complete 4.2 – Reading from a file

Exercise 4.1: Solution

Given this list:

```
>>> characters = ['Frodo Baggins', 'Samwise Gamgee',
'Gandalf', 'Aragorn II', 'Legolas Greenleaf', 'Meriadoc
Brandybuck', 'Peregrin Took']
```

Write code that takes the list, and writes its contents (one on each line) to the file `tolkien.txt`.

```
>>> with open('tolkien.txt', 'w') as file:
...     for name in characters:
...         file.write(f'{name}\n')
```

Exercise 4.2: Solution (Incorrect)

```
>>> with open('tolkien.txt') as file:
...     characters = file.readlines()
...
>>> characters
['Frodo Baggins\n', 'Samwise Gamgee\n', 'Gandalf\n',
'Aragorn II\n', 'Legolas Greenleaf\n', 'Meriadoc
Brandybuck\n', 'Peregrin Took\n']
```

What happened?

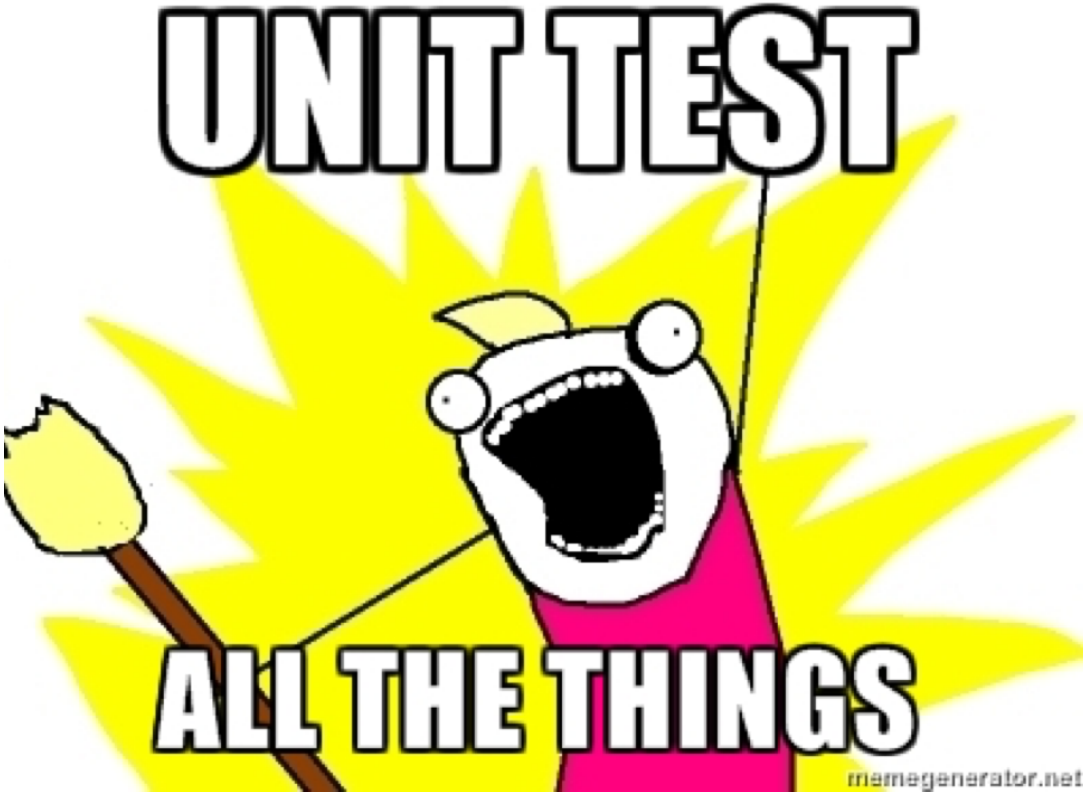
Exercise 4.2: Solution (Correct)

Use the text file we made right now, read from the file `tolkien.txt` and store each line in a list `characters`.

```
>>> characters = []
>>> with open('tolkien.txt') as file:
...   for line in file:
...       characters.append(line.strip())

>>> characters
['Frodo Baggins', 'Samwise Gamgee', 'Gandalf',
'Aragorn II', 'Legolas Greenleaf', 'Meriadoc
Brandybuck', 'Peregrin Took']
```

Better.

Testing the code



Testing the code

- Why test?
  - Assures correctness of the program under specific conditions
  - Thinking of testing while coding makes the coder write code that is better designed
  - Helps you think about edge cases (e.g. What if user tries to delete a file that isn’t there? What if a function that takes mutable data is given an immutable type?)

Choosing Test Cases

Category	Description/Examples
Size	For collections of data (lists/tuples/dicts) • empty, 1 item, small interesting case, several items
Dichotomies	• even/odd numbers • vowels/no-vowels • positive/negative • Empty/full • Etc.
Boundaries	If function behaves differently around boundaries in data, test **below** boundary, **at** the boundary and **above** the boundary
Order	If function behaves differently based on order in a list, vary order in argument.

Doctest vs. Unit Test

- Doctest
  - Informs others on how to expect your function to be used/the edge cases they may encounter
- Unit test
  - Able to run tests in a separate file which allows you to run more without worrying about cluttering the docstring

Let’s test this code

- even.py

```
def is_even(num: int) -> bool:
    """Return True if num is even.
    """

    return num % 2 == 0
```

Let’s test this code - Doctests

```
def is_even(num: int) -> bool:
    """Return True if num is even.

    >>> is_even(2)
    True
    >>> is_even(3)
    False
    """
    return num % 2 == 0

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

Doctest tests the example in the docstring

Include this to run doctest

Exercise 5.1: Testing the Code – Doctests

- Complete 5.1 – Doctests on the exercise sheet

Exercise 5.1: Solution

- Two companies tie:

```
>>> result = most_employees({'Walmart':['Trish', 'Bob'],
'Subway':['Joe', 'Anne']}
>>> result.sort() # ensures consistent order of the results
>>> result
['Subway', 'Walmart']
```

- One company:

```
>>> most_employees({'Walmart':['Trish', 'Bob']}
['Walmart']
```

Any others you thought of?

Getting Help

- Don’t spin your wheels. Come talk to us!
- Your Instructors’ Office Hours:
  - Arnamoy: Monday 11-1, BA3129
  - Danny: Monday/Tuesday/Thursday 3-5, BA2230
  - AbdulAziz: Wednesday 11-1, BA2230
- There’s lots of help in the Help Centre:
  - every weekday 2-6, BA 2230
- Your Course Discussion Board