COMP 1510 Winter 2022

Programming Methods

Week 09: Testing, dictionaries, and iteration

Agenda for week 9

1. exploring the standard library
2. math module
3. random numbers
4. constants
5. boolean expressions and, or, not
6. short-circuiting
7. repetition (looping) with while
8. sentinel values
9. infinite loops
10. breaking out of loops
11. loops and user input
12. floats and rounding
13. errors, syntax and semantics
14. testing
15. disjointed equivalency partitions and coverage
16. automated testing
17. unit testing
18. assertions
19. unit testing examples
20. dictionaries
21. iteration
22. Iterables and iterators
23. itertools and zip()
24. using enumerate() instead of range
25. ranges vs iterators vs views

25 February 2022

STANDARD LIBRARY

Modules

- *Programmers often write code in more than just a single file*
- We collect logically related functions in files called modules
- These collections can be imported for use in a program that requires that code
- We do this because we hate reinventing the wheel!
- A module is a .py file containing Python code that can be used by other modules or scripts.

Modules II

- A module can be used by being imported
- Importing a module makes the definitions inside it available
- Once a module is imported, any object defined in that module can be accessed using dot notation
- (PS: functions are objects too!)
- For example, in the time module there is a function called sleep( ) which accepts a number and suspends the program for that many seconds:

```
import time
time.sleep(3) # suspends execution for 3 dramatic seconds.
```

Style alert

- Put import statements at the top of our files
- Import statements are *case sensitive*
- Remember that when a module is imported, its code is executed
- Make sure all your code is inside functions!

Modules III

- Separating code into different modules makes management of larger programs simpler
- For example, a simple Tetris-like game might have:
  - a module for input (buttons.py)
  - a module for descriptions of each piece shape (pieces.py)
  - a module for score management (score.py), etc.
- The Python standard library is a collection of useful pre-installed modules

MATH MODULE

Math module

- I like math (have you noticed?)
- There is a non-trivial amount of math in programming:
  - Statistics
  - Predictive analysis
  - Machine learning
  - AI
  - Quantitative trading
  - Graphics (polygon mesh processing and geometric mesh modelling)
- Python has a math module

Math module documentation

https://docs.python.org/3/library/math.html

Check out some of those functions!

1. ceil( )
2. floor( )
3. fabs( )
4. pow( )
5. trunc( )

RANDOM NUMBERS: EXPLORING THE RANDOM MODULE

Let's try it out together!

More useful modules (we will use many!)

1. *string*
2. *copy*
3. *pydoc*
4. doctest and unittest
5. *pprint*
6. *statistics*
7. *getpass*
8. itertools
9. unittest.mock
10. argparse
11. http
12. http.client
13. http.server
14. sys
15. time
16. typing
17. timeit
18. os
19. json
20. csv
21. zipfile
22. difflib
23. filecmp
24. os.path
25. secrets
26. datetime
27. subprocess
28. webbrowser
29. numpy
30. pandas
31. matplotlib
32. re
33. and so very many more!

CONSTANTS

Some variables are not, in fact, variable!

- Variables tend to 'vary' (*groan*)
- Sometimes we want a variable to remain **constant**
- In many languages we can use a keyword like *constant* or *const* or *final*
- If we write code in that languages that tries to assign a new value to the constant variable, the compiler or interpreter will complain and disallow it
- Python has no such thing – Python lets us make our own rules
- Python developers tend to code carefully and use ALL_CAPS for a constant's identifier
- Constants tend to be defined at the top of the module after the import statements and before all the functions

A Python hack we can use for constants

We can do this, too:

```
def avogradro_constant():
    return 6.02e23
```

Which we can use like this:

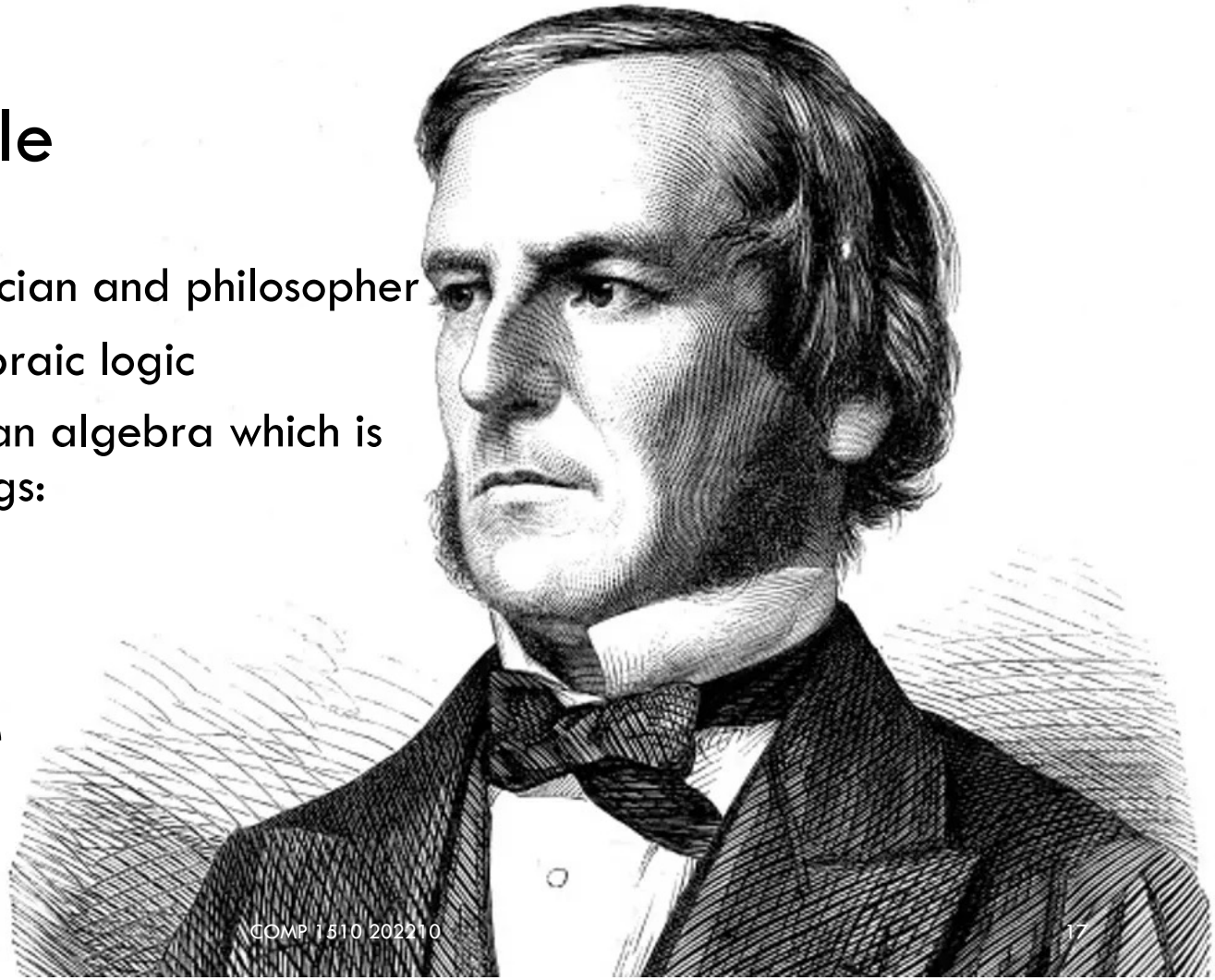
```
result = 98/ avogradro_constant()
```

BOOLEAN EXPRESSIONS

George Boole

- English mathematician and philosopher
- Specialist in algebraic logic
- Developed Boolean algebra which is all about two things:

1. True
2. False



Comparison operators (review)

We have written if statement conditions that use one of Python's **comparison operators,** which all return True or False:

Operator	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Remember these examples?

```
a = 4
b = 0
c = (a > b)
print(c) # Prints True
```

```
5 == 5
True
```

```
0 == 4
False
```

```
‘6.6’ == 6.6
False
```

```
‘A’ < ‘B’
True
```

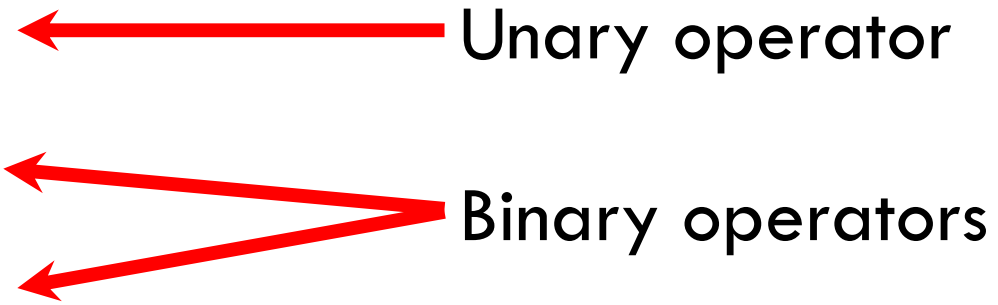
```
‘a’ > ‘z’
False
```

```
‘hello’ != ‘hi’
True
```

Introducing logical operators

Boolean expressions can also use the three following logical operators in Python:

1. not
2. and
3. or



Logical not

- Also called logical negation or logical complement
- If **a** is True, then **not a** is False
- Logical expressions like this can be shown using a truth table (you will learn about these in your math class):

a	not a
True	False
False	True

Logical or and logical and

- The logical and expression is true if both a and b are true, else it is false:

```
value = (4 > 3) and (0 == 0)
```

- The logical or expression is true if either a or b is both are true, else it is false:

```
value = (4 > 3) or (0 == 5)
```

Truth table (quick review for you!)

- A truth table shows us all the possible true false combinations
- Since **and** and **or** are both binary operators, there are four possible combinations:

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Complex expressions

- We can construct expressions that use multiple logical operators
- Be reasonable about this
- Clarity is paramount (after correctness, of course)

```
passed = (exam > 50.0) and (quizzes > 50.0 or
         assignments > 50.0 or chocolates == True)
```

Leaving ASAP aka short-circuiting!

- There is an order of operations (just like PEDMAS for arithmetic):
  1. NOT
  2. AND
  3. OR
- Boolean **and** and **or** are also ***short circuited***
- If the left operand is sufficient to determine the result, the right operand is not evaluated:

```
paid = ontime or (late and stayed-late)
goal_met = (raised > 1000) and (cost < 500)
```

Self-check: step 1

```
sales = 19
employees = 7
grade = 'a'
```

Which of the following is/are true?

1. `(sales > 16) and (sales < 25)`
2. `(sales > 16) and (employees > 10)`
3. `(sales > 16) or (employees > 10)`
4. `not (employees > 10)`
5. `not (sales > 16)`
6. `not (grade == 'a')`

Self-check: step 2

- Write Boolean expressions to represent the following:
  1. If days is greater than 30 and less than 90
  2. If maximum_cars is between 0 and 100
  3. If number_stores is between 10 and 20, inclusive
  4. If number_dogs is 3 or more and number_cats is 3 or more
  5. If either age_years is greater than 10 or age_years is less than 18. Use "or". Use > and < (not >= and <=). Use parentheses around sub-expressions.
  6. If num is a 3-digit positive integer, such as 100, 989, or 523, but not 55, 1000, or -4.

REPETITION WITH WHILE

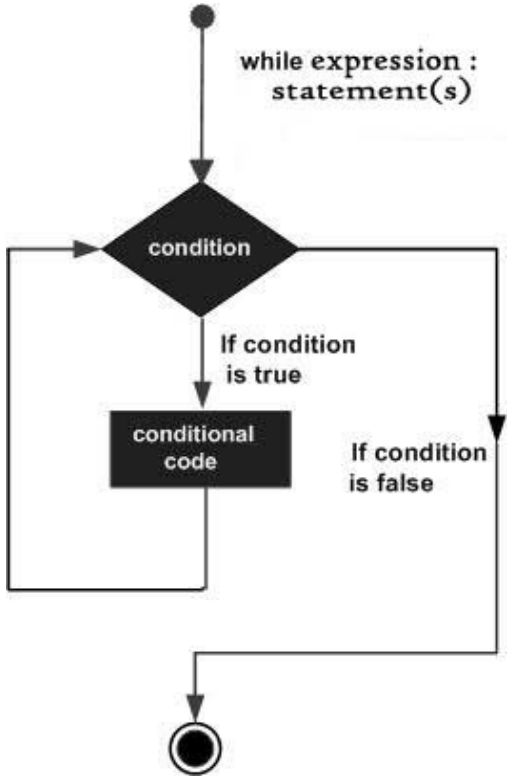
Introducing the while loop OMG finally Chris

- A **for loop** executes a block of code once for each item in a collection
- A **while loop** runs as long as, i.e., *while*, a condition is true

```
number = 1
while number <= 5:
    print(number)
    number += 1
```

Anatomy of a while loop

```
while (Boolean expression):
    # indent me
    conditional code
```



There are lots of way to use the while

We can use a counter:

1. Initialize an integer counter outside the loop (give it an initial value)
2. check in the guard condition that the counter is still within ‘bounds’
3. Execute some code in the loop
4. Remember to increment (or decrement) the counter inside the loop so that we get closer to the condition.

Example

What will this print?

```
count = 0
while count < 5:
    print(”I’m not using count inside the loop”)
    counter += 1
```

SENTINEL VALUES

There are lots of way to use the while

We can use a Boolean Flag:

1. Set a Boolean outside the loop (give it an initial value)
2. check in the guard condition that the Boolean is still True/False
3. Execute some code in the loop
4. Remember to include code in the while loop that eventually changes the Boolean.

Example

What will this print?

```
continue_looping = True
while continue_looping:
    # do some stuff
    user = input("Enter Q to quit: ")
    if user.upper() == "Q":
        continue_looping = False
```

There are lots of way to exit the while

We can use nearly anything:

```
user_input = ""
while user_input != "quit":
    # do some stuff
    user_input = input("a meaningful prompt")
```

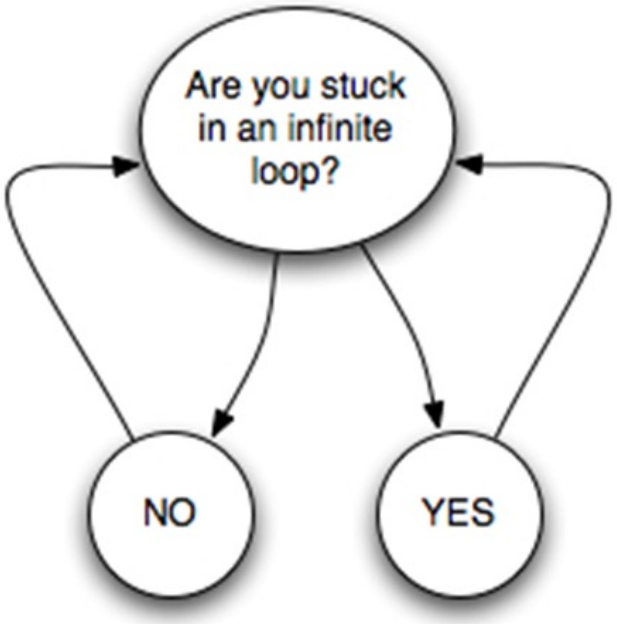
This is a sentinel value. It keeps watch, like a sentinel!

INFINITE LOOPS

Infinite loops

- If we do not eventually fail the guard condition, the while loop will never, ever end
- We call this an infinite loop
- For example:

```
count = 0
while count < 5:
    print(”This is my life now”)
```



BREAKING OUT OF LOOPS

I shouldn't be telling you this, but…

- There are two slightly controversial commands in programming
- Python has them
- They are:
  1. break
  2. continue

Break

- We can use break to exit a loop immediately!
- It exits the loop without:
  - Checking the guard condition
  - Executing any more code in the loop after the break statement
- We can use this with a for loop to exit the loop once we find something
- But this can make the flow of control difficult to follow
- *The rule is: do what makes the code easy to understand*

Example

```
while True: # or for _ in rounds:
    choice = input(“helpful prompt”)
    if choice.lower() == “quit”:
        break
    # do some stuff
```

Continue

- The continue statement returns to the beginning of the loop
- The rest of the code in the loop is ignored

```
index = 0
while index < 10:
    index += 1
    if index % 2 == 0:
        continue
    print(index, end=" ")
```

We can use the while with sequences

- For loop is effective for looping through a list
- We can use a while loop to modify a list as we work through it
- We can:
    1. Move items from one list to another
    2. Remove all instances of a specified value from a list
    3. Fill a dictionary with user input.

Moving items from list to list

```
list_one = [ 1, 2, 3, 4, 5 ]
list_two = []
while list_one:
    element = list_one.pop()
    list_two.append(element)
```

Removing all instances from a list

```
list_one = [ “Sith Lord”, “Sith Lord” ]
while “Sith Lord” in list_one:
    list_one.remove(”Sith Lord”)
    print(“Later, alligator!”)
```

LOOPS AND USER INPUT

Filling a list with user input

```
responses = []
continue_game = True
while continue_game:
    name = input("ask for name")
    option_choice = input("prompt")
    responses.append(option_choice)
    repeat = input("Continue? Y or N")
    if repeat.lower( ) == "n":
        continue_game = False
```

FLOATS AND ROUNDING

Integers can be very, very large

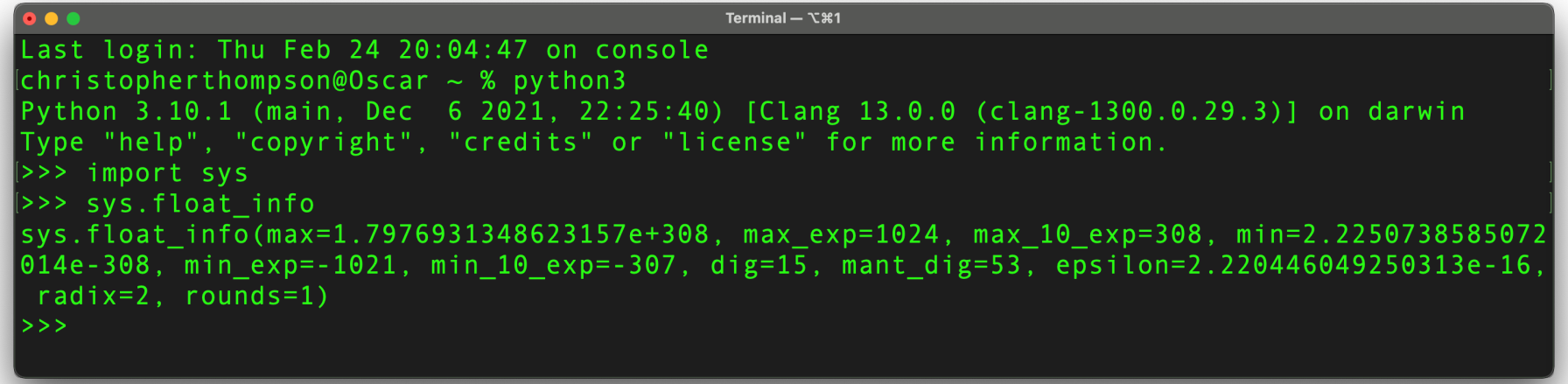
- In Python and the rest of the universe ints are whole numbers
- We can create very, very, very large ints in Python
- In Python 3, we say that the **int type is unbounded**
- Integers in Python can be as large as we want, as long as there is enough space in memory to store them
- (so effectively, the size of an int is unlimited).

But floats are not unbounded!

- Floats are implemented by the interpreter
- Usually implemented using the double floating point type in C (the language used to write the interpreter)
- We say that float type objects in Python have a **limited range of values** that can be represented.

What can we represent in floating point?

- Import the sys module
- And then print out the result from sys.float_info:



More details about floats (math review)

- Floats are generally represented in scientific notation:

$$a \times 2^b$$

- a is called the mantissa
- b is called the exponent
- The number of digits that the mantissa and the exponent contain are limited
- This limits the precision with which floating-point numbers can be stored and the magnitude of the numbers
- Because the exponent can have only a limited number of digits, a float cannot be arbitrarily **large or tiny**
- Because the mantissa can only have a limited number of digits, a float cannot be arbitrarily **precise** (i.e. it cannot have an arbitrary number of digits after the decimal)

Float vs int

- Without loss of generality…
- **Floating-point** types should be used to represent quantities that are measured, such as distances, temperatures, volumes, weights, etc.
- **Integer** types should be used to represent quantities that are counted, such as numbers of cars, students, cities, hours, minutes, etc.

Comparing values

- In Python we compare values with **==**
- We check if two arguments refer to the same object with **is**
- We can do this with:
  - Ints
  - Booleans
  - Strings
  - Lists
  - Tuples…
- What about floats? How do we compare floats?

Floating point numbers

- Represented as base 2 binary fractions
- Compare:
  - Base 10: 0.125 is 1/10 + 2/100 + 5/1000
  - Base 2: 0.001 is 0/2 + 0/4 + 1/8
- Most decimal fractions cannot be represented exactly as binary fractions
- Compare:
  - Base 10: 1/3 is 0.333333333333…
  - Base 2: 1/10 is 0.0001100110011001…
- We call this representation error

Quick fix for floats I of III

- We cannot always represent floating point numbers precisely with binary
- We can account for that using a TOLERANCE
- If the absolute value of the difference between two floats is less than a TOLERANCE that we define, then we can assume the two floats are **equal enough**.

Quick fix for floats II of III

- Define a constant called **TOLERANCE**
- Instead of checking if two floating point numbers are equal, check that the absolute value of the difference is less than TOLERANCE
- The value of TOLERANCE depends on your program:

```
TOLERANCE = 0.0001
a = some floating point number
b = some floating point number
if abs(a – b) < TOLERANCE:
    print('a is close enough to b')
```

Quick(est) fix for floats III of III

Use the isclose function from the math module!

1. Use one (or both) of the optional keyword parameters:
    1. rel_tol describes the ± range as a percentage of the larger operand
    2. abs_tol describes the ± range as a positive floating point number.
2. When numbers get very very small, describing the range as a percentage becomes mathematically ridiculous
3. So we can use the abs_tol if we expect numbers to be tiny
4. Python gives us flexibility and control over the choice.

ERRORS, SYNTAX, AND SEMANTICS

Syntax and semantics

- The ***syntax rules*** of a language define how we can put together symbols, reserved words, and identifiers to make a valid program that the interpreter can execute
- The ***semantics*** of a program statement define what that statement means (what it does)
- A program that is syntactically correct is not necessarily logically (semantically) correct
- A program will always do what we tell it to do, not what we meant to tell it to do

https://docs.python.org/3/reference/lexical_analysis.html

Python is interpreted (mostly)

- There are FOUR steps involved in executing a Python program. Our file is processed by a few programs:
  1. **Lexer** generates a stream of tokens using these* rules:
  2. **Parser** reads the tokens using syntax rules
  3. **Compiler** creates Python bytecode file
  4. **Interpreter** consumes the bytecode and run the program.
- The first three steps produce Python bytecode, which is interpreted by the Python interpreter during the fourth step

Errors can occur during any step

A Python program can have three types of errors:

1. The parser may find ***syntax errors***
2. The interpreter may encounter ***run-time errors***
3. A program may run to completion but may generate incorrect results (***logical errors***)

Syntax errors

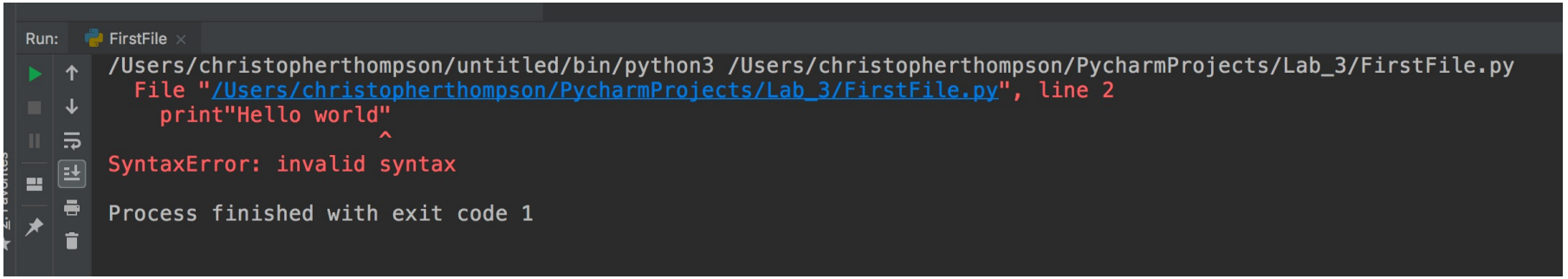
```
print"Hello world"

def add(a, b):
return a + b

x =
```

These will be underlined and will usually produce some sort of warning in the right-hand margin of the PyCharm code editor

Syntax errors

- *Assertion*: If we try to run a program that contains syntax errors, nothing will happen
- *Proof*: Put these two lines of code in a file and try to execute it. Does anything get printed? That is, does the first line get executed?

```
print("Hello world")
print"Hello world"
```



Runtime errors

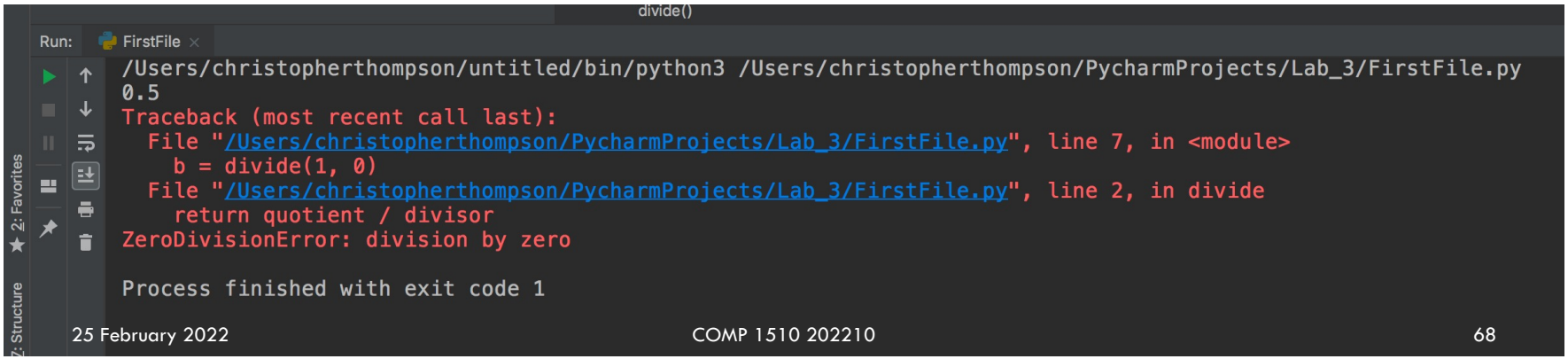
- The most famous run-time error is to **divide by zero**
- Dividing by zero rains **chaos and fury upon the earth**
- Try something like the following program:

Try me

```
def divide(quotient, divisor):
    return quotient / divisor


a = divide(1, 2)
print(a)
b = divide(1, 0) # I dare you. Make my day.
print(b)
```

Run-time errors are called *exceptions*

- Even if a statement is syntactically correct, it can still cause an error when it is executed
- Errors detected during execution are called *exceptions* in Python (because something exceptional has occurred)
- Most exceptions result in error messages like this:



Logic errors (semantic errors)

- The program executes and does not crash
- Output is “weird” and incorrect

```
def divide(quotient, divisor):
    return quotient + divisor # You wot, mate?


a = divide(1, 2)
print(a) # 3?!
b = divide(1, 0)
print(b) # Will this code ever execute?
```

We must deal with all errors!

- **Early** errors (**syntax** errors) are easy to spot thanks to our lexer, parser, and compiler
- **Later** errors (**logic** errors and **run-time** errors)
  - Also known as **BUGS**
  - The lexer, parser, and compiler cannot help with these
  - Interpreter is not always helpful with its messages
  - Some logical errors have no immediately obvious manifestation
- Aside: commercial software is rarely error free (sigh)

TESTING

Testing = verification + validation

- *Verification*: "Did we build the program correctly?"
  - Discover situations in which the behavior of the software is incorrect or undesirable
- *Validation*: "Did we build the correct program?"
  - Demonstrate to the developer and the customer that the software meets its requirements

Five time-honoured testing axioms

1. Testing **can't** show that bugs **don't** exist.
2. A bug is a bug only if it's observed. Bugs that haven't been found yet are simply **undiscovered bugs**.
3. It is virtually impossible to find subtle bugs in a program while it still contains major bugs.
4. We cannot fix all the bugs in a software application .
5. Specifications and requirements are a **moving target**; features will be added, removed, or changed during development.

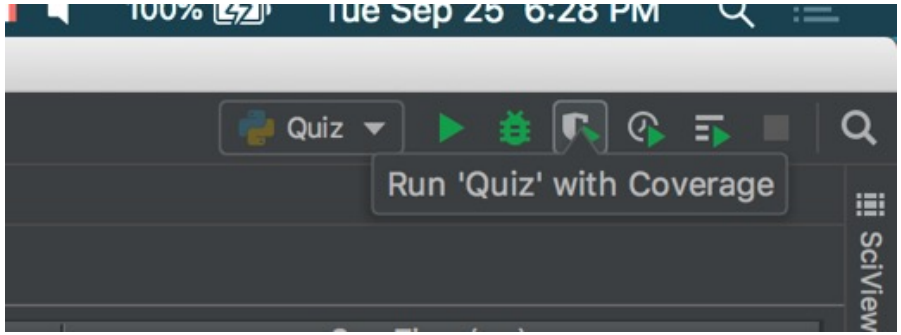
Testing fundamentals (start here!)

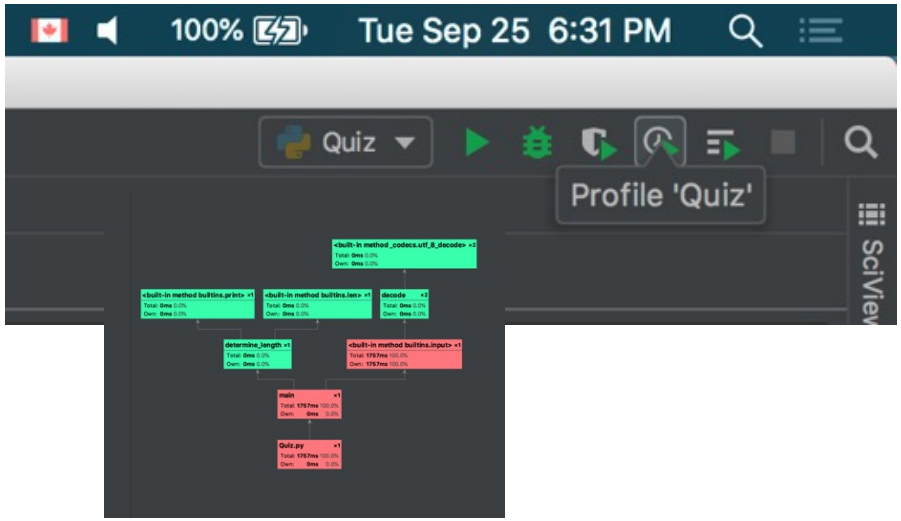
- Start with the basics:
  1. *Format and organize* your code
  2. *Document* your code
  3. Apply what you're learning about variables, scope, functions, etc. – you want the fewest moving parts possible!
- By default, PyCharm analyzes all open files and highlights all detected code issues in the editor
- On the right side of the editor, you can see the analysis status of the whole file

Remove dead code!

- We talk about code coverage. How much of the code you wrote is actually used during execution?
- **Run your program "with coverage"**
- **Watch this!**

- We profile code too. We take a snapshot so we can examine what has happened.
- **"Profile" your program and examine the Call Graph.**
- **Watch again!**





Some testing fundamentals

1. Understand what the test subject should do – its ***contract*** – and consider what it consumes…
2. We can categorize tests into positive tests and negative tests:

+ testing the subject using valid data YES!

- testing the subject using invalid data NO!

Testing fundamentals

3. Start by testing *boundaries*:
   - Zero, One, Full.
   - Max/Min
   - Just inside/just outside boundaries
   - Typical values
   - Error values

Create DISJOINTED EQUIVALENCE PARTITIONS

DISJOINTED EQUIVALENCY PARTITIONS

Disjointed Equivalence Partitions

1. *Coverage*: Identify what types of inputs are going to be processed in a similar way (equivalent behaviour)
2. *Disjointedness*: Each group of inputs is a partition that does not overlap with other groups of inputs
3. *Representation*: Each test should exercise only one equivalence partition

Math people: it's a little like the Cartesian product

Equivalency partition example 1

- Suppose we create a function that is supposed to correctly process an integer between 100 and 999, i.e., a 3-digit integer
- Equivalence partitions:
  1. Less than 100 (we don't care!)
  2. 100 to 999
  3. Greater than 999 (we don't care!)
- Choose 3 tests:
  1. 100 (the boundary)
  2. 500 (some random value in the middle)
  3. 999 (another boundary)

Equivalency partition example 2

Suppose I have a function called `validate` which accepts a pair of integers `x` and `y`. The function returns `False` if the product of the integers is positive, and `True` if the product of the integers is negative.

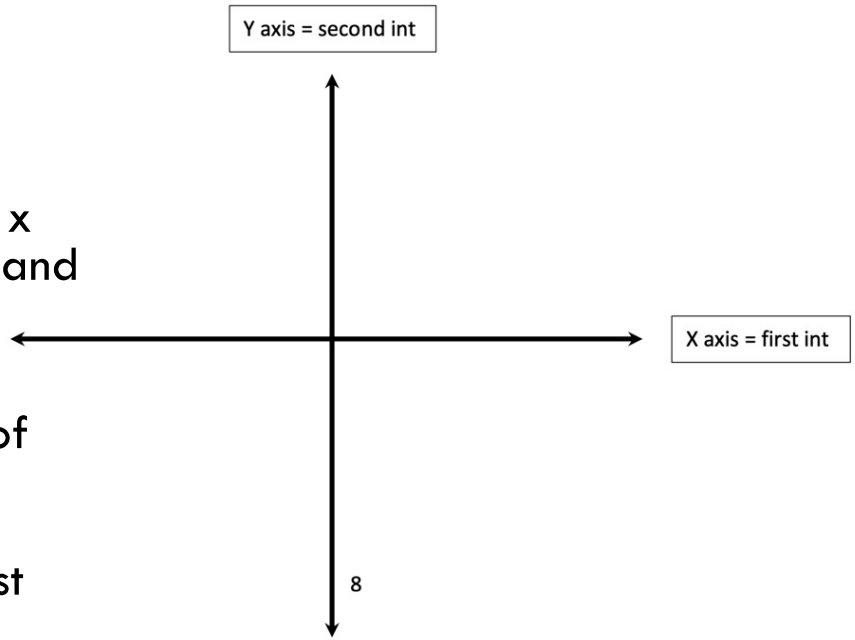
We can break this down into four **explicit cases**:

1. If the first and the second integer are both positive, the function returns False
2. If the first and the second integer are both negative, the function returns False
3. If the first integer is positive and the second number is negative, the function returns True
4. If the first integer is negative and the second number is positive, the function returns True

And finally, for the purpose of this function, zero is considered positive.

Equivalency partition example 2

- Draw and label the four disjointed equivalence partitions in this number graph representing the two integers passed to validate as arguments. Let x represent the first integer parameter, and let y represent the second integer parameter
- When testing this function, which sets of values should we use? That is, which points on the graph would you test? Label each set of points you would test (see example). Hint: there are 9.



UNIT TESTING

Doctesting is very useful and we will use it

- We will use doctests for most of our functions
- We will also use **unit testing**
- Doctests have a different use case from unit testing
- **Unit tests are more detailed and language agnostic**

What is a unit test

What: a piece of code that exercises a very small, specific area of functionality of a unit of code (for 1510 a unit is a function).

How: A unit test invokes a particular function or method in a particular context, i.e., testing a value from a disjointed equivalency partition

Example: add a middlesome value to a sorted list, then confirm that the list size has grown by one OR that the list is still sorted.

Goal: isolate important parts of program and prove they are as correct as we can.

Unit tests

- A **unit** is a testable part of an application
- Each **unit** of an application is tested in isolation:
  1. Function
  2. Method
  3. Behaviour
  4. Composite components with defined interfaces used to access their functionality…

Each unit test must:

1. Assemble
2. Act
3. Assert

For each unit test, consider:

1. Starting state
2. Inputs to which the unit responds
3. Expected results or end state

Introducing unittest

- The unittest module is the **gold standard** Python test module
- Its **API** is similar to JUnit (Java), CPPUnit (C++), nUnit (.NET)

API = Application programmer interface

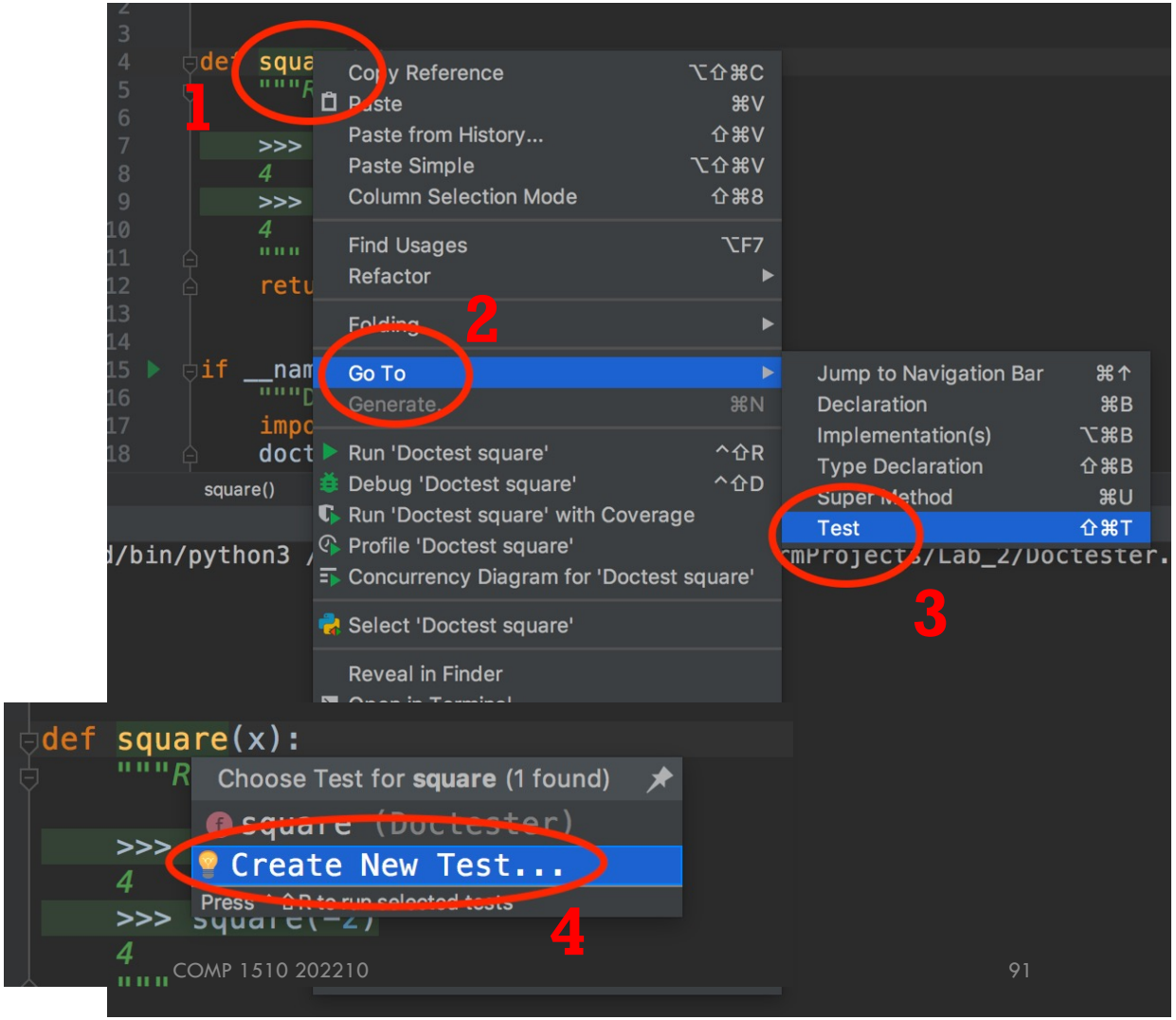
- The basic building block of unit tests is a test case
- A test case is a test function that sets up a single scenario and asserts that our function responds correctly
- We can apply this knowledge to all programming

unittest https://docs.python.org/3/library/unittest.html

- The unittest module and unit testing in general is a sophisticated and complex tool
- We are going to explore it bit by bit all term
- We will **start with the very basics**
- This will make you a top programmer (I'm not kidding)

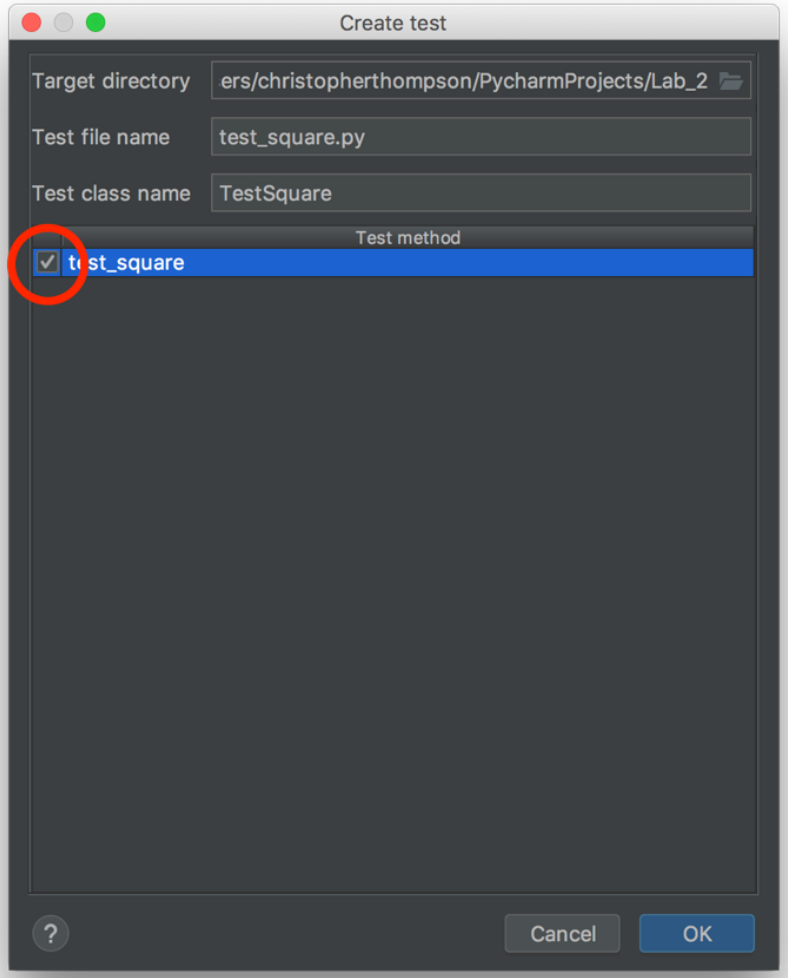
The easiest way to do this is:

- Right click the function we want to use
- Select Go to > Test
- In the dialog, select Create New Test



The Create Test window opens…

- Leave everything alone except:
  - Check your function
  - Then press okay



What did we just do?

- We used PyCharm to **automatically** create a unit test module
- Almost all IDEs can do this for their respective languages
- Automation makes testing easier and (sometimes) fun
- We created a unit test file
- Examine:
    1. Name of the file
    2. The first line
    3. The rest of the code.

What should we add to test our code?

- A collection of unit tests is a collection of test functions inside a class (sort of like a module, we will examine it later this term!)
- Each test is a function that contains one assertion and in rare cases more than one assertion
- All the assertions from all the tests we write for a function must test all the disjointed equivalency partitions
- *The unit test functions must all begin with the word test*

Test coverage (things we will consider)

1. **Order**. If a function behaves differently when values appear in different orders, identify these orders and test each of them
2. **Boundaries**. If a function behaves differently around a particular boundary or threshold, test exactly that boundary case.
3. **Dichotomies**. A dichotomy is a contrast between two things like empty/full, even/odd, positive/negative, alphabetical/notalphabetical, etc. *If a function deals with two or more different categories or situations, make sure you test all of them*.

ASSERTIONS

Let's add something like this:

```
def test_square(self):
    self.assertEqual(4, module_name.square(2))
```

- Import your unittest module at the top of the unit test file
- Don't worry about the self stuff (we will cover self in a few weeks)
- Use a single assertion statement inside each test function
- There is a list of assertions we can use on the next page
- We ASSERT that the function reacts the way the function is supposed to

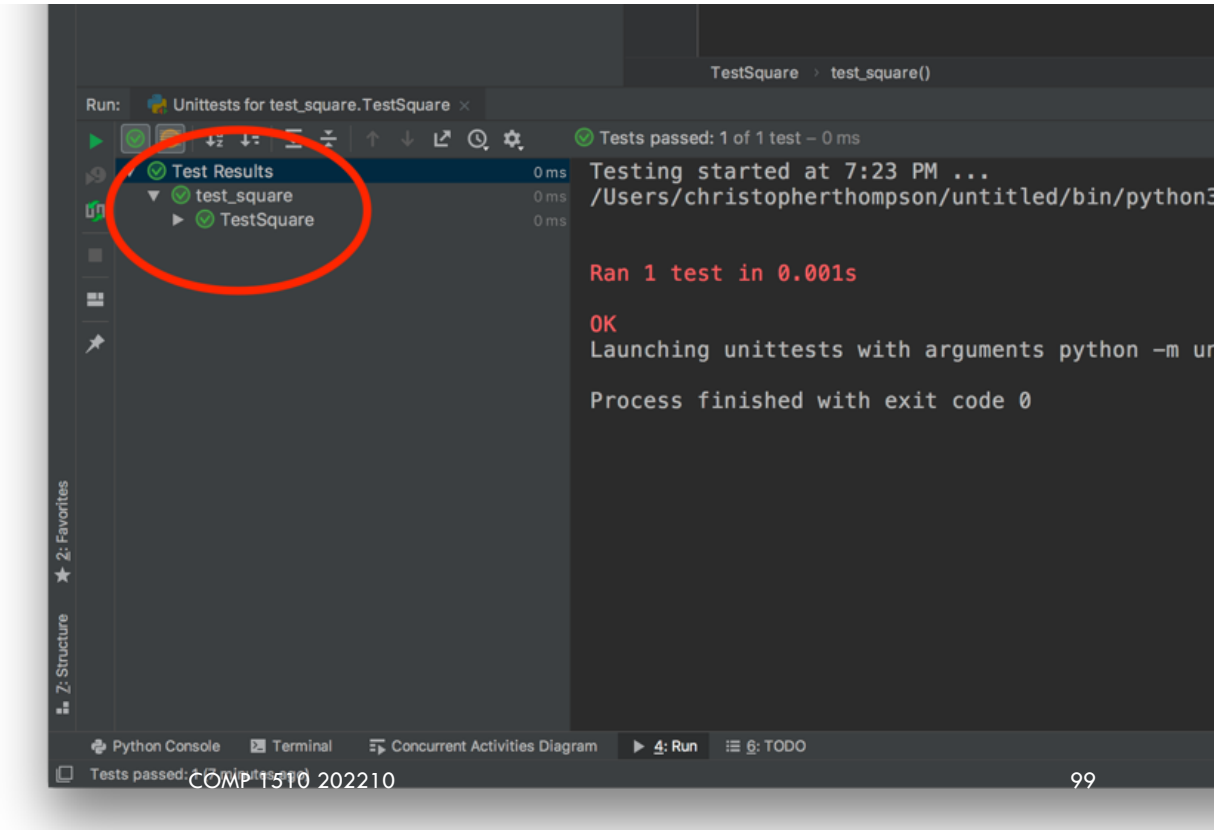
Which assertions can we make?

https://docs.python.org/3/library/unittest.html#assert-methods

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

How do we run the unit tests?

- Right-click the unit test class or the unit test function
- Click a green arrow in the margin
- Choose Run | Run 'UnitTests…
- We want to see this:



UN PETIT MOT ABOUT TESTING

We will use doctests to:

1. show users how to use our functions
2. describe the consequences of invoking our functions
3. illustrate how our functions react to different kinds of input
4. We will create a few doctests for each function
5. Some functions cannot be adequately tested with doctests, though…

We will learn how to use "unit tests" to:

1. test functions that cannot be adequately tested with doctests
2. demonstrate that postconditions are true when a function is correctly invoked by a user who meets the preconditions
3. assert that an expected error occurs when a function receives certain data
4. generate mock objects that represent externalities so we can control what our function receives from external helper functions
5. redirect standard output and test what functions print
6. create predictable random numbers.

DICTIONARIES

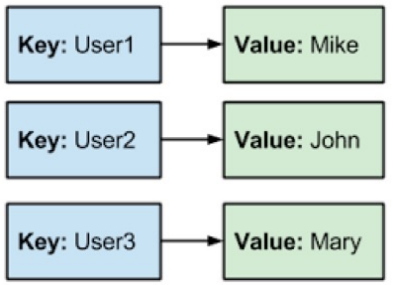
The Python dictionary

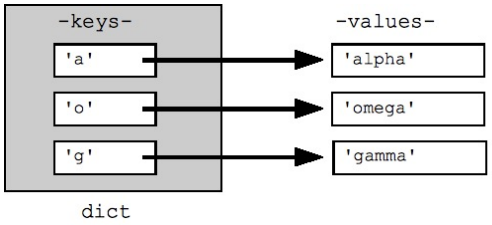
- In Python, a dictionary is a collection of key-value pairs
- Each key (the first thing in the pair) is joined to a value (the second thing in the pair)
- We use curly braces to delineate the dictionary
- Each key and value pair is connected by a colon **:**
- Each pair is separated from other pairs by commas **,**

```
dictionary_one = { 'name' : 'Lwaxana Troi' }
dictionary_two = { 1 : 'COMP1510', 2 : 'COMP2522', }
dictionary_three = { 'Picard' : 'SP-937-215' }
empty_dictionary = { }
```

Key-value pairs

- Sometimes keys are called attributes
- K-V pairs are everywhere in programming:
  - Tables whose contents are arranged and accessed by key
  - Query strings in URLS (example.com/lang?name=python)
  - Element attributes in markup, i.e., HTML
- A dictionary is also called:
  - Hash table
  - Associative array





testEnvironment

Key	Value
protocol	https
host	apigee.com
port	443
version	/v1

encryptedStuff

Key	Value
username	*****
pasword	*****
oauth_key	*****
aws_key	*****

Keys must be unique

- In Python **keys must be unique** in a dictionary
- Each key must be associated to one value
- *Values do not have to be unique*
- Many keys can be associated to the same value

```
>>> will_this_work = { 'key' : 'value', 'key': 'value2', }
>>> will_this_work
{'key': 'value2'}
```

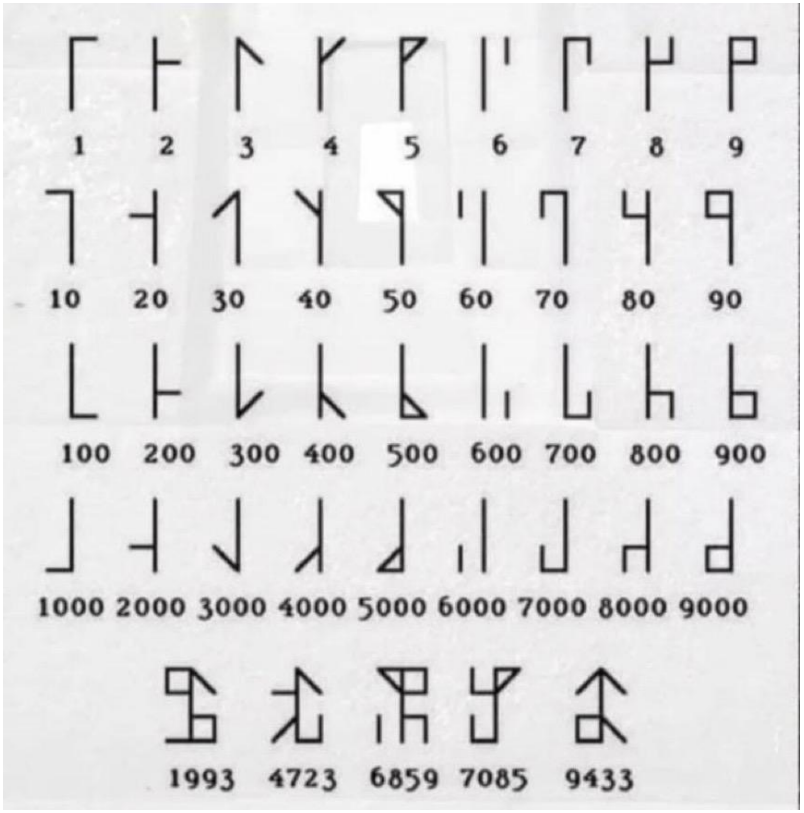
What can we use for keys?

- We may NOT change the key once it is in the dictionary
- Strings, numbers, tuples (*use anything immutable*)
- The keys in a dictionary do not have to be the same type, either!

```
>>> my_dictionary = {}
>>> my_dictionary['value'] = 100.0
>>> my_dictionary[1] = 1
>>> my_dictionary[2] = 2
>>> my_dictionary[2.5] = 'hello'
>>> my_dictionary
{'value': 100.0, 1: 1, 2: 2, 2.5: 'hello'}
```

Challenge: what's this?

```
>>> my_dictionary[()] = []
>>> my_dictionary
{(): []}
>>> my_dictionary[()] = 'hello world'
>>> my_dictionary
{(): 'hello world'}
```



Accessing values in a dictionary

- Use square bracket notation: value = dictionary[key]
- Instead of an index (like a list) we provide the key:

```
>>> grades = { 'A00123456' : 80.0, 'A00987654' : 90.0, 'A00112358' : 100.0 }
>>> grades['A00123456']
80.0
>>> your_grade = grades['A00112358']
>>> your_grade
100.0
>>> my_grade = grades['A00555555']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A00555555'
```

Growing a dictionary

- The dictionary is a dynamic structure (it grows and shrinks as we add and remove key-value pairs):

```
>>> my_character = { 'name' : 'Boaty McBoatface' }
>>> my_character
{'name': 'Boaty McBoatface’}
>>> my_character['occupation'] = 'water bard'
>>> my_character
{'name': 'Boaty McBoatface', 'occupation': 'water bard'}
```

Dictionaries are mutable

- We can modify the values in a dictionary:

```
>>> my_character = { 'name' : 'Boaty McBoatface' }
>>> my_character
{'name': 'Boaty McBoatface’}
>>> my_character['name'] = 'Mister Splashy Pants'
>>> my_character
{'name': 'Mister Splashy Pants'}
```

```
my_character = { 'name' : 'Mister Splashy Pants'}
my_character['speed_mps'] = 1.0
my_character['x_coordinate'] = 0.0
my_character['y_coordinate'] = 0.0
my_character['direction'] = 'south'

if my_character['direction'] == 'south':
    y_increment = -1
elif my_character['direction'] == ‘north':
    y_increment = 1
my_character['y_coordinate’] += y_increment
print(my_character)
```

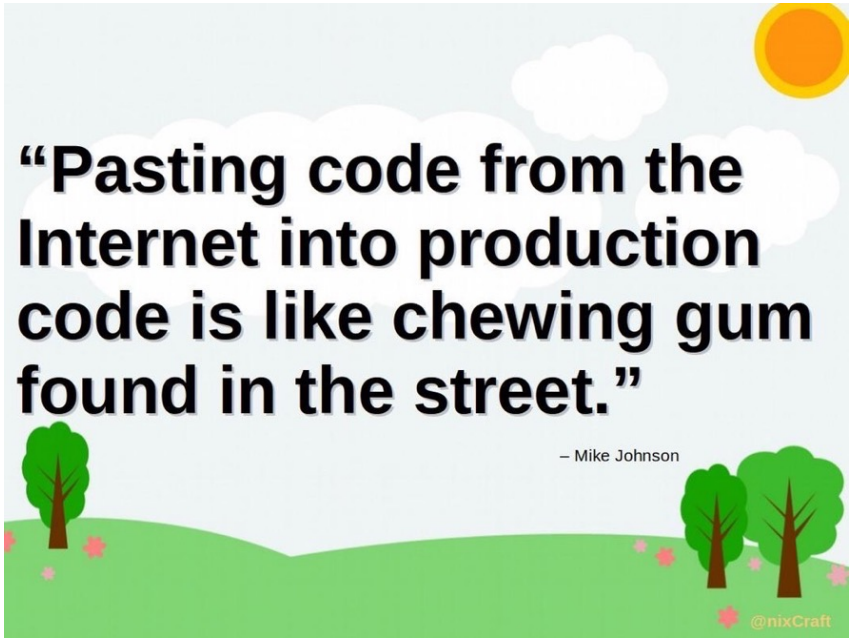
Dictionaries are mutable

- We can delete key value pairs too
- Use the del statement

```
>>> my_dict = { 1: 168000, 2: 166000, 3: 167000, 4: 168000, 5: 165000, 6: 166000}
>>> my_dict
{ 1: 168000, 2: 166000, 3: 167000, 4: 168000, 5: 165000, 6: 166000}
>>> del my_dict[6]
>>> my_dict
{ 1: 168000, 2: 166000, 3: 167000, 4: 168000, 5: 165000}
```

How do we loop through dictionaries?

- We can loop through a dictionary in 3 different ways:
  1. Through its key-value pairs
  2. Through its keys
  3. Through its values.



1. Looping by key-value pair

- The items( ) method returns a “view” of key-value pairs
- The for-loop temporarily assigns the address of each pair in two variables we must provide:

```
>>> character = { 'str': 15, 'int':13, 'wis':10}
>>> for attribute, value in character.items():
...     print(attribute, value)
str 15
int 13
wis 10
```

2. Looping by key

- The keys( ) method returns a list of keys
- This is the default behaviour when working with a dictionary
- The following are equivalent:

```
>>> for key in character.keys():
...     print(key, character[key])
...
str 15
int 13
wis 10
```

```
>>> for key in character:
...     print(key, character[key])
...
str 15
int 13
wis 10
```

We can sort the keys before looping too

- Suppose we have a dictionary that maps names to favourite programming languages
- We can do this:

```
fave_langs = {“Chris”: “Python”, “Keanu”: “Java”,
              “Kim”: “JavaScript”}
for name in sorted(fave_langs.keys()):
    print(name.title() + “likes” + fave_langs[name])
```

3. We can loop through the values

- Maybe we are primarily interested in the values stored in the dictionary
- We can use the values( ) method, like this:

```
for language in fave_langs.values():
    print(language.title())
```

Looping through a dictionary: a summary

- Use the dictionary's *items( )* method to loop through key-value pairs
  - The items( ) method returns a "view" of key-value pairs
  - The for-loop stores each pair in two variables which we must provide
- The *keys( )* method returns a "view" of keys
  - (This is default behaviour when working with an associative array)
- If we are primarily interested in the values stored in the dictionary we can use the *values( )* method

ITERATION

it·er·a·tion

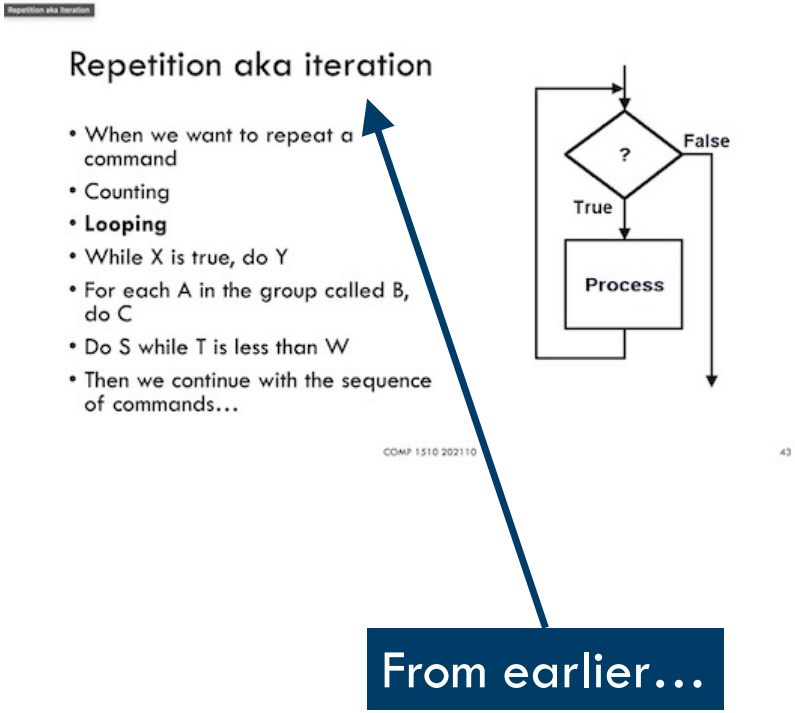
/ˌidəˈrāSH(ə)n/

Noun

- the repetition of a process or utterance.
- repetition of a mathematical or computational procedure applied to the result of a previous application, typically as a means of obtaining successively closer approximations to the solution of a problem.
- a new version of a piece of computer hardware or software.

Iteration

- Recall that we define repetition to be a synonym for iteration
- In Python (and other languages), ***iteration*** implies that we are looping over the members of a collection, i.e., a list, a range, etc.
- Iteration is fundamental to data processing



We use **iterators** to **iterate** over **iterables**

- The for-loop seems very easy to use
- It is actually a lightweight wrapper around some very cool code
- The for-loop uses abstraction to protect us from complexity
- The Python for-loop iterates over an iterable
- An iterable is an object capable of returning its members one by one
- ***Q: How do I know if something is a "Python iterable"?***
- ***A: I can iterate over its members with a Python for-loop*** 🙃

The for-loop is hiding a LOT from us!

- There are two important built-in functions in Python:
  1. iter( )
  2. next( )
- When we pass an 'iterable' to the iter( ) function, the iter( ) function will return a new special iterator object
- We must pass that special iterator object to the next( ) function again and again and again
- Each time we do, the next( ) function will get the "next" element from the iterator!

How does an iterator work?

- We say that the iterator "streams" the contents of the container
- Calling next( ) returns successive items in the stream
- When no more data is available, a *StopIteration exception* is generated and dealt with
- When this happens, the iterator is exhausted and cannot be used again. It must be discarded. Like a toxic relationship.
- An iterable produces a fresh iterator object each time we:
  1. Pass it to the iter( ) function
  2. Use it in a for-loop.

So what kind of things can be iterable?

- Anything in Python can be iterable, as long as it can return the elements it contains one by one
- It must be a data type that implements one of two special methods that get invoked by the iter( ) and next( ) functions:
  1. __iter__( ) creates an iterator object that can iterate over the objects in the container
  2. __getitem__( ) provides the means to access objects in the container using keys composed of integer or slice objects

Chris, in Python what *specifically* is iterable?

1. Sequences like lists, strings, tuples
2. Dictionaries
3. Generators and virtual sequences created from functions like range( ), ***enumerate****( ), ***zip****( ), reversed( ) and methods like the three dictionary methods items( ), values( ), keys( ), etc.
4. Sets (coming soon)
5. File objects (coming later this term!)

* We're about to see what these do!

And what about the special iterator object…

- The special iterator object returned by the __iter__( ) method must implements two more methods:
  - __next__( ) returns the next available item or raises StopIteration
  - __iter__( ) which returns a reference to itself

- In COMP 2522 and 3522, you will probably implement iterables and iterators
- For now, being able to intelligently describe what is happening under the hood and how the for-loop abstracts away the need to use iter( ) and next( ) is enough!

Example (for fun) and a question?

```
my_fruity_tuple = ("apple", "banana", "cherry")
my_fruity_iterator_object = iter(my_fruity_tuple)
print(next(my_fruity_iterator_object))
print(next(my_fruity_iterator_object))
print(next(my_fruity_iterator_object))
```

Q: What happens if I try to `print(next(my_fruity_iterator_object ))` again?

a) Nothing
b) The universe as we know it will cease to exist
c) The Python runtime interpreter will raise a StopIteration exception
d) You can avoid this unnecessary stress by using a for-loop, Chris

In 1510 we will employ abstraction!

The for-loop abstracts away the following steps:

1. Create an iterator from the given iterable
2. Repeatedly access the next member of the iterable from the iterator
3. Execute the desired action
4. Halt the looping if we receive a StopIteration exception when accessing the next member

Remember programming is all about layers of abstraction

Where does Python use iterators?

Whenever the interpreter needs to iterate over the members of an object x, it automatically calls iter(x) for us:

1. For-loop
2. List, dict, and set comprehensions (coming up)
3. Tuple unpacking, i.e., x, y = 10, 20
4. Looping over text files line by line (soon!)
5. Unpacking variable length parameter lists passed with * in function calls (soon!)

Iteration: developers love to iterate

- We loop through collections of data a lot!
- Sometimes we need to:
  - Add, accumulate, or aggregate values
  - Generate streams, sequences or arrangements of data
  - Tabulate statistics
- Most languages have some built in tools for efficient iteration and stream generation
- Python includes the itertools module

The itertools count function

- https://docs.python.org/3/library/itertools.html#itertools.count
- The itertools.count(start=0, step=1) function makes an iterator object that "streams" an infinite sequence of evenly spaced values
- We generally don't use it by itself because the sequence never ends
- Try this (please don't hate me):

```
import itertools
for number in count(0):
    print(number)
```

What does this snippet do?

```
evens = itertools.count(0, 2)
even_numbers = []
for _ in range(10): # Ask me about the _ please ☺️
    even_numbers.append(next(evens))
print(even_numbers)
```

We can use floating point numbers with count

```
import itertools

for value in count(0.25, 0.5):
    print(value)
```

What does this snippet do?

```
count_with_floats =
    itertools.count(start=0.5, step=0.75)
float_sequence = []
for _ in range(5):
    float_sequence.append(next(count_with_floats))
print(float_sequence)
next_value = next(count_with_floats)
print(next_value)
```

The itertools.cycle( ) function

The itertools.cycle(iterable) function makes a "generator object" that "streams" the individual values in the iterable and returns them (cycling through the values):

```
choices = []
boolean_generator = itertools.cycle([True, False])
for _ in range(20):
    choices.append(next(boolean_generator))
print(choices)
```

ENUMERATE( )

Enumeration with enumerate( )

- This is a built-in function
- It takes one or two parameters
- The first parameter is an iterable object
- The optional second parameter is an integer that represents a starting counter value
- Enumerate returns an iterable object that we can pass to the for-loop
- As we loop through each element the iterable object gives us, we must assign the return value to two variables
- Here is another example:

Enumerate: to assign a number!

```
alphabet_ords = list(range(65, 91))
for ascii_value, letter in enumerate(alphabet_ords, 65):
    print(ascii_value, chr(letter))

def even_items(iterable):
    for index, value in enumerate(iterable):
        if index % 2 == 0:
            print(index, value, end=", ")

even_items("Christopher Thompson")
```

The itertools.permutations( ) function

The itertools.permutations(iterable) function accepts an iterable and returns all the possible re-arrangements of the elements:

```
primary_colours = ['cyan', 'magenta', 'yellow']
permutations = list(itertools.permutations(primary_colours))

print("There are %d permutations: " % len(permutations))

for sequence_number, permutation in enumerate(permutations, 1):
    print("%d:\t%s" % (sequence_number, permutation))
```

The itertools.combinations(iterable, r) function

```
size = 2
combos = list(itertools.combinations(
    ["ginger", "allspice", "cumin", "mint"], size) )
print("There are %d combinations of size %d:" %
    (len(combos), size))
for seq_number, combo in enumerate(combos, 1):
    print("%d:\t%s" % (seq_number, combo))
```

ZIP( )

The count( ) function is useless without a friend

We can use it with zip( ) to generate a sequence of numbers:

```
pairs = []
names = (“Victoria”, ”Edmonton”, “Regina”, “The Peg”)
for pair in zip(itertools.count(1), names):
    pairs.append(pair)
print(pairs) # What gets printed here?
```

What's zip( )?

- If we have two sequences of the same length, we can iterate over both at the same time
- We visit members in corresponding indices
- We call this *iteration in parallel*
- zip( ) is cool. It's a lazy generator. It doesn't zip everything up right away…
- During each loop, it generates ONE tuple containing the next values from each iterator (so it does this "on demand").

Example problem and a few solutions…

```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names] # SNEAK PEEK 😍

longest_name = None
max_letters = 0

// Our code goes here

print(longest_name)
```

Example solution 1 🙁

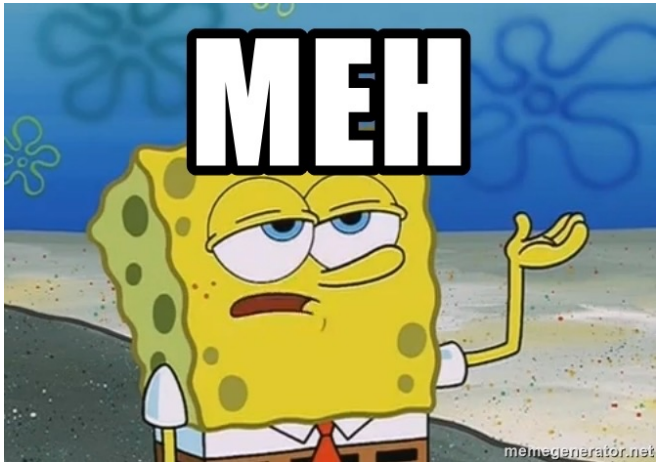
```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]

longest_name = None
max_letters = 0

for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count

print(longest_name)
```

This is visually noisy.

Using indices to reach into into letters and names makes this hard to read.

Example solution 2 😐

```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]

longest_name = None
max_letters = 0

for i, name in enumerate(names):
    count = letters[i]
    if count > max_letters:
        longest_name = name
        max_letters = count

print(longest_name)
```



This is marginally better

Less indexing makes it easier to read

But it's still noisy

Example solution 3 using zip( ) 🙂


```
names = ['Cecilia', 'Lise', 'Marie']
letters = [len(n) for n in names]

longest_name = None
max_letters = 0

for name, count in zip(names, letters):
    if count > max_letters:
        longest_name = name
        max_letters = count

print(longest_name)
```

Best!

No indices!

So short!

Easy to read and understand!

This is parsimony!

VIEW VS RANGE VS ITERATOR

When a list is not a list but in fact a view

Remember dictionaries?

```
meals = {'bfast':'egg', 'lunch':'poke',
         'dinner':'spinach'}

keys = meals.keys()
values = meals.values()
entries = meals.items()
```

When a list is not a list but in fact a view

These methods return *views*, not lists:

```
>>> print(type(keys))
<class 'dict_keys'>

>>> print(type(values))
<class 'dict_values'>

>>> print(type(entries))
<class 'dict_items'>
```

What is a view

- https://docs.python.org/3/library/stdtypes.html#dict-views
- A virtual sequence (like the range object we use)
- We use it for iteration
- Provides a dynamic 'view' of the entries
- Dynamic: when the dictionary changes, so does the view

- Views support three functions:
  1. *len*(dictview) returns the number of entries in the dictionary
  2. x *in* dictview returns True if x is in the underlying dictionary's keys(), values(), or items()
  3. *iter*(dictview) returns an iterator over the view

Some view examples

```
>>> dishes = {'eggs': 2, 'sausage': 1,
              'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # Use a view for iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504
```

```
>>> # keys and values are iterated over
in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and
reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']
```

That's it for week 09!

This was a lot. I know. Thank you for working so hard.
Please spend time working on A4 this weekend!