# Welcome!

COMP 3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 1: PYTHON FUNDAMENTALS

# Agenda

1. Introduction
2. Logistics and Expectations
3. Intro to Python
4. Comments and docstrings
5. PEP-8 Style Guide
6. Strings, ASCII and Unicode in Python
7. Formatting output (str.format, f-strings, conversion specifiers)
8. Mutability in Python
9. Sequence types
10. Lists
11. Tuples
12. Dictionaries
13. Iteration and views

# A Bit About Me…

Jeffrey Yim (call me Jeff)

**Office Hours:**

**SW2 - 127**

**Monday**: 12:30 PM – 3:30 PM
**Tuesday:** 1:30 PM – 3:30 PM

Email:

- **jyim3@bcit.ca**
- **Subject line [COMP3522]**

# Me

Education:

- Started at BCIT! w/Pascal
- Queen's University
- Bachelor's/Master's Computer Science

Interests:

- Game development, technology
- Unity

Favorite language

- C#

# Industry games










# iOS games


ROGUE


Custom
EPIC
Posters




BLITZ




DRAW!
LITE



LITE


Custom
EPIC
Posters+


TYPING WITH
VOLCANOES


...

# Let’s Get Through Some Administrative Stuff First

# A Bit About The Course…

- Object Oriented Programming II (I know it's a bit on the nose…)
- A **6 Credit Course**
- Specifically, understanding how to architect and design modular object oriented code. You will (eventually) write more realistic OO code that models abstract relationships.
- This course is all about Dependencies and Coupling (we will talk more about this in the coming weeks). The everlasting dilemma that all developers face.
- Design Patterns to solve common problems
- Probably THE most important programming course

# Class Schedule

**Lecture 1 (All Sets)**
Tuesday, 8:30 AM – 10:30 AM
SW09 - 110

**Lecture 2 (All Sets)**
Friday, 8:30 AM – 10:30 AM
SW05 - 1850

# Lecture Breakdown

1. Traditional lecture
2. Examine code – have small discussion
3. In-class activities
4. In-class quizzes
5. Questions always welcome
6. Give me visual feedback
    1. nodding
    2. shaking head

# Assessments

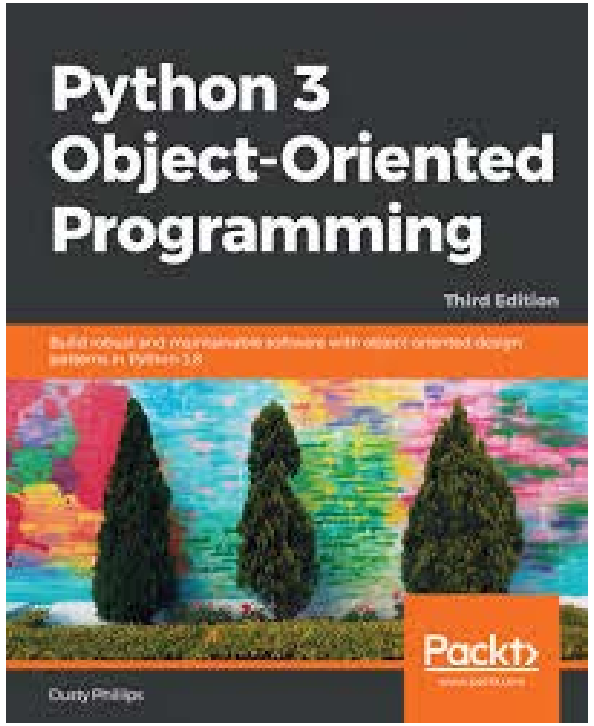
- Quizzes 10%
- Labs 10%
- Assignments 30%
- Midterm 20%
- Final exam 30%

To Pass:

Submit all assignments
50% overall
50% weighted average between the Midterm and Final Exam

Please aim higher (obviously)!

# Textbook (Optional)




Philips, Dusty. (2018) ***Python 3 Object Oriented Programming*** *3rd Edition*. Packt Publishing;

- The text is available as an eBook via the BCIT Library and is a great supplemental read.
- There is a real shortage of tutorials and materials on advanced OOP in Python. This is a good one.

# Let's Lay Down Some Ground Rules…

- Be on time. Let me know if you can't make it. Attendance will be taken each class. (There is an attendance requirement).
- Be professional. Let's make our class a safe space with no judgements.
- Assignments should be submitted on time. Late submissions will be penalized (barring emergencies, medical reasons, or any such special circumstance).
- Plagiarism and cheating **won't be tolerated**. If you submit work that is not yours, it **NEEDS** to be accredited. Assignments are to be submitted individually unless they explicitly require you to work in groups. If you are found in infringement of these policies, this will result in a grade of **ZERO**! (BCIT Plagiarism Policy: https://www.bcit.ca/files/pdf/policies/5104.pdf )
- That being said, don't hesitate to discuss concepts introduced in class and help each other understand them. If you are in doubt, err on the side of caution. Ask me if you're not sure!

# Let’s Lay Down Some Ground Rules…

- You are responsible for your own learning. Ask questions, come to office hours, reach out for assistance if you are falling behind and are stressed out.
- Phones should be on Silent and put away.
- Support your peers and know when to ask for help! Your class mates are your support network and will be your contacts in the industry in the future. Be nice ☺
- Again, ask questions if in doubt! Interrupt the class or raise your hand. There are no stupid questions, and I will reiterate, this is a judgement free safe space. You are here to learn. If you want to chat privately or are outside of class, shoot me an email!
- Most importantly, **have fun!** Enjoy the learning process.

# Slack

With Slack!

Our Slack workspace is called

**https://join.slack.com/t/comp3522-bby-wint2020/signup**

*Use Slack for all communication. If it is something important, send me an email as well.*

Sign up and send me a private message right now!

# Finally, Let’s Get Started!

# Intro to Python

# Why Python?

Modern (released in 1991, updated every few months)

**Easy** to learn

Lots of **support** on the net

Large and comprehensive standard **library**

Supports multiple programming paradigms like **OOP**, functional, and procedural

It's an **interpreted** (not compiled) language

Python is dynamically typed and strongly typed

We say that batteries are included: there is a large standard library with GREAT documentation (see python.org)

# Python and Comp 3522

This is a second course in OOP

The concepts you learn here will be largely language agnostic

We can teach this course in C++ or Java or any other object oriented language

But Python is in the ascendant and is a leading language that implements the OOP Paradigm

According to the TIOBE index, Python is now the third most popular programming language!

https://www.tiobe.com/tiobe-index/ *

# What Can You Do With Python?

Linux Scripting & Administration

2D and 3D Modelling

Image Processing (Effects and Manipulation)

UI Applications

Data Analysis and by extension Big Data

Game Development

Machine Learning

- Face detection
- Computer Vision
- Neural Networks

# Python Has Libraries and Frameworks

Standard Library

Django for website development

Numpy for machine learning

TensorFlow for Artificial intelligence

SciPy for scientific computations

PyQt for cross-platform GUI development

UnitTest (PyUnit) for unit testing our code

Scrapy for scraping information from web sites

Requests for HTTP fun

# Hello World

# My first Python program

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

- Seriously, that's it!
- Notice there's no semicolon ;
- Python executes whatever code that's in the file, even without a main function. Code is interpreted top to bottom
- Where's the main function???

# My first Python program

- Need to add some code to the bottom of the file

```
if __name__ == '__main__':
    main() #main is a function we'll write
```

- __name__ is a special Python variable
- Its value is automatically set by Python
  - __name__ is set to '__main__' if the code begins running from a script
  - __name__ is set to the module filename if the code started running from a different script

# My first Python program

```
def main():
    print('hello world')


if __name__ == '__main__':
    main() #calls main function to print hello world
```

- This is the complete code to print out hello world, with a main function
- What this code is saying is "if we're running code starting from this script, call a main function"

# My first Python program

```
def jeff():
    print('hello world')


if __name__ == '__main__':
    jeff() #does the same thing as the previous slide
```

- Notice there's nothing special about the naming of the main function
- I've changed the function to jeff and it behaves the same

# Intro to Variables and Functions

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

num1 = input('Enter a number')
num2 = input('Enter a second number')
print(add(num1,num2))
```

- Function bodies begin after the :
- Notice how there are no curly braces
- Code sections are separated by indentation. Notice the indent before 'return a + b'
- That's not just for readability, python requires indentation to separate code sections

# Intro to Variables and Functions

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

num1 = input('Enter a number')
num2 = input('Enter a second number')
print(add(num1,num2))
```

Missing indent!

```
    return a + b
    ^
IndentationError: expected an indented block
```

- Compiler error if there's no indent for function. This applies for if/else, for, while etc
- Think of where you'd normally have curly braces in Java, and that's where you'd indent

# Intro to Variables and Functions

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

num1 = 5
num2 = 10
print(add(num1,num2))
```

- Python is a dynamically typed language.
- Notice how there's no type (int) in front of num1. Types are inferred

# Intro to Variables and Functions

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

num1 = 5
num2 = 10
print(add(num1,num2))
```

- Same as parameters in a function, notice how types are not specified

# Intro to Variables and Functions

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

num1 = 5
num2 = 10
print(add(num1,num2))
```

- Can add : type after the parameter to indicate type
- But this is just for programmer intention/readability and it doesn't change functionality
- Can still pass in types other than int into function and it will work

# Intro to Variables and Functions

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

num1 = 5
num2 = 10
print(add(num1,num2))
```

- Functions begin with def
- def is used in place of void or return types you're used to in Java
- No need to specify a function is void if it doesn't return anything, or a return type if it does return something

# Intro to Variables and Functions

```
def add(a, b) -> int:
    return a + b

num1 = 5
num2 = 10
print(add(num1,num2))
```

- Can add -> type after the function header to indicate return type
- But this is just for programmer intention/readability and it doesn't change functionality
- Function can still return non-int types

# The if statement

```
if_stmt ::=  "if" expression ":" suite
             ("elif" expression ":" suite)*
             ["else" ":" suite]
```

# Branching: if

A branch in a program is **only taken if an expression's value is true**

This is known as an if-branch:

```
def calculate(salary, earned_bonus):
    if earned_bonus == True:
        salary = salary * 1.2
    return salary
```

# Branching: if-else

An if-else structure has two branches:

1. The first branch is executed if the testing expression is true
2. The second branch is executed if the testing expression is false

```
def positive(number):
    if number < 0:
        return false
    else
        return true
```

# An example

```
password = input("Enter your password: ")

if password == "1234" :
      print("Unacceptable password")
else:
      print("Thank you")

print("End of program")
```

# Another example

Getting the maximum of two numbers:

```
def max(first, second):
    if first >= second
        max_value = first
    else
        max_value = second
    return max_value
```

# Be careful with your indentation!

```
def positive(number):
    if number < 0:
        print(”Negative”)
        return False
    else:
        print(”Positive”)
        return True
```

We can insert a **group (block) of statements** after if and else

# Multi-branch if-else

```
if expression1:
    # Statements that execute when expression1 is true
    # (first branch)
elif expression2:
    # Statements that execute when expression1 is false
    # and expression2 is true
    # (second branch)
else:
    # Statements that execute when expression1 is false
    # and expression2 is false
    # (third branch)
```

# Another example

```
password = input("Enter your password: ")

if password == "1234":
	print("Unacceptable password")
elif password == "abcd”:
	print("Dude, come on")
else:
	print("Thank you")

print("End of program")
```

# Nested if-else statements

A branch's statements can include any valid statements, including another if-else statement

```
if sales_type == 2:
   if sales_bonus < 5:
     sales_bonus = 10
   else:
      sales_bonus = sales_bonus + 2
else:
   sales_bonus = sales_bonus + 1
```

# We can use multiple sequential ifs

Sometimes we can use multiple if statements in sequence

**Each if statement is independent**

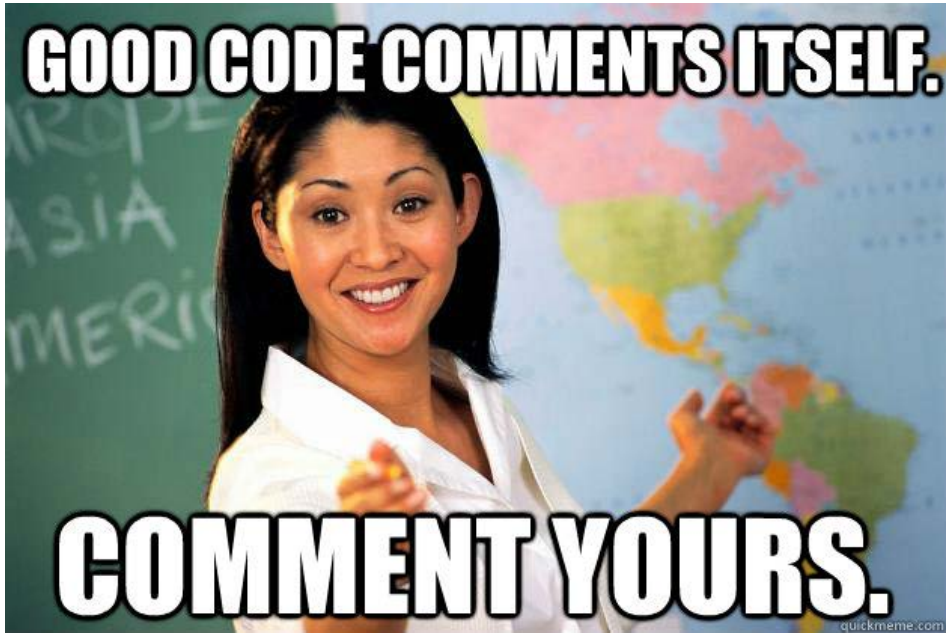
More than one can execute:

```
user_age = int(input('Enter age: ‘))
if user_age >= 16:
   print('You are old enough to drive.')
if user_age < 25:
   print('Enjoy your early years.')
if user_age > 25:
   print('Most car rental companies will rent to you.')
```

# Comments and Docstrings

# # Good Comments & Readable Code

- Why do we comment?
- We strive to make our code self-documenting by using descriptive identifiers for functions, variables, etc.
- Good code should self explanatory (right?)
- Python Comments
  - In line comments that start with #
  - Docstrings



# Docstrings

## A special kind of comment

## A string literal that describes a chunk of code

Implemented as the first statement inside a:

- Module (file)
- Function
- Class
- Method definition.

Used to create “official” documentation

Wrap with triple-quotes to span multiple lines

Python can use this to generate documentation

# Docstring Example

```
def add_ints(a, b):
    """
    Return the sum of the two arguments.
    :param a: an int
    :param b: an int
    :precondition: a must be an int
    :precondition: b must be an int
    :return: the sum of the ints
    """
    return a + b
```

# Docstring Formatting

Ensure you use a phrase that ends in a period.

Describe the function's effect as a command, i.e., "'''**Do** this,''' or """ **Return** that."""

Do not describe the function, i.e., do not write "~~**Returns**~~ the pathname…"

A multi-line docstring must start with a one-line summary, followed by a blank line, followed by the details of the description.

Suppose we have a program called hello.py. The first statement in the file must be a triplequote wrapped string, which will become the hello.py module's docstring when the file is imported and used by another project.

```
>>> import mymodule
>>> help(mymodule)
>>> help(mymodule.my_function)
```

# Styling Your Code

FASHION POLICE

# PEP 8 Style Guide

https://www.python.org/dev/peps/pep-0008/

- The defacto accepted standard for formatting your Python code.
- At some point everyone here should go through it.
- There are a lot of rules but it eventually becomes second nature.
- Another website that summarized the PEP 8 guide (An easier read): https://realpython.com/python-pep8/

## Why does this matter?

# PEP 8 Style Guide

Consistency

Readability

Parsing

Efficient and Ease While Coding

Collaboration

| Type | Naming Convention | Examples |
| --- | --- | --- |
| Function | Use a lowercase word or words. Separate words by underscores to improve readability. | function, my_function |
| Variable | Use a lowercase single letter, word, or words. Separate words with underscores to improve readability. | x, var, my_variable |
| Class | Start each word with a capital letter. Do not separate words with underscores. This style is called camel case. | Model, MyClass |
| Method | Use a lowercase word or words. Separate words with underscores to improve readability. | class_method, method |
| Constant | Use an uppercase single letter, word, or words. Separate words with underscores to improve readability. | CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT |
| Module | Use a short, lowercase word or words. Separate words with underscores to improve readability. | module.py, my_module.py |
| Package | Use a short, lowercase word or words. Do not separate words with underscores. | package, mypackage |

There a lot more rules that I won’t reiterate here. Check out the links!

# Strings

# Python String

A **string** is an immutable sequence (or string) of Unicode (UTF-8) codepoints

We can bind a string object to a variable

A **string literal** is the value of a string

We create a string literal using double- **or** single-quotes:

May contain letters, numbers, spaces, or symbols like @ or #.

"BCIT CST grads are the best"

'BCIT CST grads make the most money'

# Python Strings are a Sequence Type

A string's letters (characters) are kept in order from first to last

A character's position in a string is called its index

The plural of index is indices

Indices start at 0

| P | Y | T | H | O | N |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

An empty Python string has zero elements and is created like this:

```
my_empty_string = ''
```

# Unicode

Python uses Unicode to represent every possible character as a unique integer, known as a **code point**

For example, the character ‘A' has the code point value of 65

Sometimes converting between a text character and the encoded code point integer is useful. (Easier to process numerical data, especially when analyzing large sets or during encryption).

The built-in function **ord( )** returns the encoded code point integer for a string of length one (a Unicode character).

The built-in function **chr( )** returns the one-character string for a code point passed as an argument.

# Unicode > ASCII

Programmers used to use **ASCII** (American Standard Code for Information Interchange)

Encodes **128** characters, mostly from English

This was fine in the early days of computing, but 128 characters is obviously insufficient now

**Unicode is a superset of ASCII** (the first 128 characters of Unicode are the ASCII characters)

We will focus on UTF-8, which uses 1 Byte for the ASCII characters, and up to 4 Bytes for each of the rest.

# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# Escape sequences

They're the ***same*** as everywhere else

Another way to print quotation marks is to wrap the string around different quotes as opposed to its content

Eg: "That's skipping an unnecessary \' ."

```
print("hello'")
print('hello"')
```

| Escape Sequence | Meaning |
|---|---|
| \' | Single quote (') |
| \" | Double quote (") |
| \t | Horizontal tab |
| \a | Bell (please don't do this) |
| \b | Backspace |
| \r | Carriage return |
| \n | New line (line feed) |

# Raw Strings

If we do not want to use escape characters we can create a raw string

A raw string is prefixed by the letter r or R

Python raw strings treat the backslash as a literal character

This is useful if we have a string that contains backslashes that should be interpreted as backslashes

Compare:

- `print(“Hello\nworld”)`
- `print(r“Hello\nworld”)`

# Some Common String Methods, Functions and Features

Try them out now!

- `“String”.title()`
- `my_string = “Comp 3522”`
- `my_string.upper()`
- `my_string.lower()`
- `len(my_string)`
- `my_string[3]`
- `my_string [0]`
- `my_string[-2]`

**Question: if we invoke these methods, does the original string change?**

# Some Common String Methods, Functions and Features

**Question: if we invoke these methods, does the original string change?**

**NO!**

# Some Common String Methods, Functions and Features

1. capitalize
2. replace
3. isalpha
4. isdigit
5. split
6. join
   - this is faster than concatenation using += because no temp strings are created
   - This is an important and fast way to concatenate strings

https://docs.python.org/3.7/library/stdtypes.html#string-methods

# Some Common String Methods, Functions and Features

1. capitalize
2. replace
3. isalpha
4. isdigit
5. split
6. join
   - this is faster than concatenation using += because no temp strings are created
   - This is an important and fast way to concatenate strings

https://docs.python.org/3.7/library/stdtypes.html#string-methods

**REMEMBER:** You can always type help(function name) to read the documentation in say iPython

Eg: `help(str.isdigit)`

# String Concatenation

What happens when you do this?

```
one_piece_cast = “Luffy”
num_members = 6 #Thats where I stopped
print(one_piece_cast + “ and “ + num_members + “others”)
```

# String Concatenation

What happens when you do this?

```
one_piece_cast = “Luffy”
num_members = 6 #Thats where I stopped
print(one_piece_cast + “ and “ + num_members + “others”)
```

NOTE: Try casting num_members to a string

# Multi-line strings

There are three ways we can do this:

1. Break lines with the \ character:

```
string_long = "This is a very long string" \
              " that I wrote to help somebody" \
              " who had a question about" \
              " writing long strings in Python"
```

# Multi-line strings

2. Use parentheses:

```
string_long = ("This is a very long string"
               " that I wrote to help somebody"
               " who had a question about”
               " writing long strings in Python”)
```

# Multi-line strings

3. Use triple quotes:

```
string_long = """This is a very long string
                 that I wrote to help somebody
                 who had a question about
                 writing long strings in Python""“
```

NOTE: PEP-8 has some thoughts about multi-line code too!

# String Formatting

# Formatting output

There are three easy ways to format output in Python

We can use:

1. the string format method
2. f-strings
3. format specifiers

It is always better to format than to concatenate! Change your habits if you tend to use ‘+’ to display data

# The str.format() Method (I prefer this personally)

The format method inserts values into strings

Replacement fields are delimited by curly braces:

```
“I earned {0} in {1}”.format(98.5, “Architecture”)
“I earned {} in {}”.format(98.5, “Architecture”)
“I earned {0} in {1} and {0} in {2}”.format(65,
“Comm”, “Math”)
“I earned {grade} in {course}”.format(grade=65,
                                      course=“Comm”)
```

# The str.format() Method

Eg: Try examples like these now!

```
grades = (98.5, 65)
“I earned {g[0]} and {g[1]}”.format(g = grades)
import math
“Pi equals {m.pi}”.format(m = math)
“Pi equals {m.pi:.3f}”.format(m = math)
```

# F-strings

f-strings are string literals preceded by an f or F (like raw strings!)

Curly braces contain expressions that can be replaced with their values

Lets you drop in variable names directly in place

Evaluated at runtime:

```
grade = 98.5
course = “C”
f”I earned {grade} in {course}”
name = “OOP 2 with Python”
F”My favourite course (so far) is {name.title()}!”
```

# A note on print()

There are different versions of the print( ) function:

1. print(‘Hello world’)
2. print(“Hello world”)
3. print(“Hello world”, end=‘ ’) #default end is newline
4. print(“I\’m ” + str(age) + “ years old”) *
5. print(“I\’m”, age, “years old”)

You can learn more here:

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

# Conversion Specifiers

Program output commonly includes the value of variables as a part of the text

A **string formatting expression** allows a programmer to create a string with placeholders that are replaced by the value of variables

This placeholder is called a **conversion specifier**

Different conversion specifiers are used to perform a conversion of a given variable value to a different type when creating a string

# Conversion Specifiers

| Conversion Specifier | Notes | Example | Output |
|---|---|---|---|
| %d | For an integer | print('%d' **%** 17) | 17 |
| %f | For a floating point number | print('%f' **%** 3.14) | 3.14 |
| %s | For a string | print('%s' **%** name) | Oliver |
| %x, %X | For hexadecimal | print('%X' **%** 17) | 11 |
| %e, %E | For scientific notation | print('%e' **%** 314) | 3.140000e+02 |

# Conversion Specifiers (floats)

```
apr = float(input('Enter APR:\n'))

# Print using a float conversion specifier
print('Annual percentage rate as a float is %f ' % apr)

# Print using a float conversion specifier and trailing %
print('Annual percentage rate as a float is %f%% ' % apr)
```

Need extra % sign for % symbol to appear in string

# Conversion Specifiers (Multiple)

**Super easy:**

name = ‘COMP 3522’

room = 655

print(‘Course %s is in %d\n’ % **(**name, room**)**) *#make sure to have enclosing brackets for multiple params*

# Mutability

# im·mu·ta·ble

/i(m)ˈmyo͞o͞dəb(ə)l/

Adjective

From the Latin roots “in” (not) and “mutabilis” (to change)

unchanging over time or unable to be changed

# String are Immutable

Writing or **altering** individual characters of a string variable is **not allowed**

## Strings **cannot change once created**

Instead, an assignment statement must be used

```
word = ‘supercalifragilisticexpialidocious’
word = word.title( )  # this allocates new memory (and un-references the previous one)
print(word)
```

# Python Immutable Types

int( )

float( )

complex( )

str( )

tuple( )

bytes( )

frozenset( )

Everything else is mutable (I think!).

# Assignment Is Not Mutation

When we assign a value to a variable, we are actually assigning the reference to that value's location in memory

We will review this in some detail later in the term

For now, we can test this like so:

```
id(2.5)
a = 2.5
type(a)
id(a) # same address as id(2.5)!
a = a + 0.0456
id(a) # Not the same – a contains the address of a new float!
```

Phew! Right, that should be enough of strings. Let’s move on to Sequences!

# Sequence Types

PROBABLY THE THING PYTHON IS MOST FAMOUS FOR

# Sequence Types

Sequence is the generic term for an ordered set

There are several distinct kinds of sequence types in Python

1. str (the string)
2. **list**
3. range
4. tuple
5. bytes

These are not the same thing.

They are stored differently and are treated differently, but they can be processed similarly

**Only <u>some</u> sequence operations are common to all**

# Common Sequence Operations

| Operation | Result |
|---|---|
| `x in s` | `True` if an item of *s* is equal to *x*, else `False` |
| `x not in s` | `False` if an item of *s* is equal to *x*, else `True` |
| `s + t` | the concatenation of *s* and *t* |
| `s * n` or `n * s` | equivalent to adding *s* to itself *n* times |
| `s[i]` | *i*th item of *s*, origin 0 |
| `s[i:j]` | slice of *s* from *i* to *j* |
| `s[i:j:k]` | slice of *s* from *i* to *j* with step *k* |
| `len(s)` | length of *s* |
| `min(s)` | smallest item of *s* |
| `max(s)` | largest item of *s* |
| `s.index(x[, i[, j]])` | index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*) |
| `s.count(x)` | total number of occurrences of *x* in *s* |

* Well almost.

Range only supports item sequences that follow specific patterns, and hence doesn't support sequence concatenation or repetition

# Mutable Sequence Operations

| Operation | Result |
|---|---|
| `s[i] = x` | item *i* of *s* is replaced by *x* |
| `s[i:j] = t` | slice of *s* from *i* to *j* is replaced by the contents of the iterable *t* |
| `del s[i:j]` | same as `s[i:j] = []` |
| `s[i:j:k] = t` | the elements of `s[i:j:k]` are replaced by those of *t* |
| `del s[i:j:k]` | removes the elements of `s[i:j:k]` from the list |
| `s.append(x)` | appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`) |
| `s.clear()` | removes all items from *s* (same as `del s[:]`) |
| `s.copy()` | creates a shallow copy of *s* (same as `s[:]`) |
| `s.extend(t)` or `s += t` | extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`) |
| `s *= n` | updates *s* with its contents repeated *n* times |
| `s.insert(i, x)` | inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`) |
| `s.pop([i])` | retrieves the item at *i* and also removes it from *s* |
| `s.remove(x)` | remove the first item from *s* where `s[i]` is equal to *x* |
| `s.reverse()` | reverses the items of *s* in place |

# Containers

A **container** is a data structure used to group related values together

A kind of compound type

A container contains references to other objects

A **list** is a container created by surrounding a sequence of variables or literals with square brackets [ ]

It's like an array, but since it's an object it has useful methods available. (It's similar to but better than Java's ArrayList)

# List

This line of code creates a new list variable my_list that contains the two items, an integer and a string:

```
my_list = [10, 'abc’]
```

A list item is called an ***element***

A list is ordered, and indexed. Just like arrays in other languages.

```
my_empty_list = []
```

A list can have many different types of elements. That is, its**heterogenous**.

```
student_data = [“A022343234”, [99,70,85.0], ‘P’]
```

# List

Lists are useful for reducing the number of variables in a program (a single list can store an entire collection of related variables)

Individual list elements can be accessed by their index

A **list's index must be an integer**.

```
>>> student_data = [“A022343234”, [99,70,85.0], ‘P’]
>>> student_data[0]
```

Try this:

```
>>> student_data[-1]
>>> len(student_data)
```

# Lists are Mutable

Let’s try some of these list methods!

```
list_name.append(value)
list_name.pop(index) # remove and use value
list_name.remove(value) # remove by value
list() (Eg: spelling = list(“ComplicatedWord”)
print(list_name)
list_name[1] = ‘new updated value’
del list_name[3]  # remove by position
```

```
list_name.sort()    # does this change the original list? Or does it return
                                a new instance?
sorted(list_name) # how does this differ from list.sort?


len(list_name)


reverse(list_name)


dir(list)
```

# Sequence Functions and Methods

**Sequence-type functions** are built-in functions that operate on sequences like lists and strings

Examples include len( ), sorted( )

**Sequence-type methods** are methods built into the class definitions of sequences like lists and strings

Examples include sort( ), append( ), clear( ), insert( ), pop( ), remove( ), and reverse( )

## What's the difference?

# Membership Operator

A common task is to determine if a container contains a specific value

Python has membership operators **in** and **not in**

These operators return true if the left operand matches the value of some element in a container

```
>>> colours = [‘red’, ‘white’, ‘cerulean’]
>>> print(‘cerulean’ in colours)
>>> print(‘magenta’ not in colours)
```

# Membership Operators

Can be used with sequence types (string, list so far)

- We can check if a list contains something
- We can also use to determine whether a string is a **substring**, or a matching subset of characters, in a larger string:
- So much easier in Python compared to say C or C++

```
request_str = 'GET index.html HTTP/1.1'
if '/1.1' in request_str:
    print('HTTP protocol 1.1')

if 'HTTPS' not in request_str:
    print('Unsecured connection')
```

# Tuples

- Think Lists, but fixed and unchanging. That is, it is **immutable**.
- A tuple is also a sequence type, supporting len( ), indexing, and other sequence type functions

```
Eg: pos_vector = (3, 7.0, -39)
```

- Not commonly used, but great when element position and data is usually fixed and unchanging. Eg: location (longitude, longitude)

# Tuples (Example)

```
parliament_hill_coords = (45.4236, 75.7009)
print('Coordinates:', parliament_hill_coords)
print('Tuple length:', len(parliament_hill_coords))

# Access tuples via index
print('\nLatitude:', parliament_hill_coords[0], 'north')
print('Longitude:', parliament_hill_coords[1], 'west\n')

# Error. Tuples are immutable
parliament_hill_coords[1] = 50
```

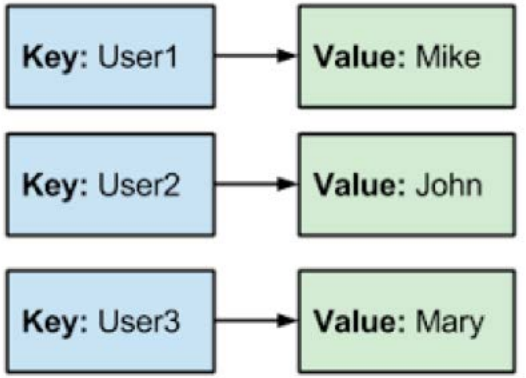
# Dictionaries

Similar to other languages, it is a **collection of key-value pairs**

Each key should be unique

```
dictionary_one = { ‘name’ : ‘Lwaxana Troi’ }
dictionary_two = { 1 : ‘COMP3522’,
                   2 : ‘COMP1712’, }
dictionary_three = { ‘Picard’ : ‘SP-937-215’ }
empty_dictionary = { }
```

# Key-Value Pairs



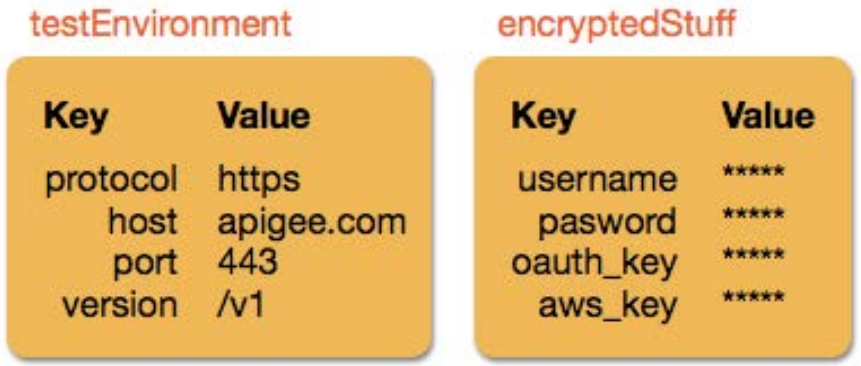

Sometimes keys are called attributes

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 languages like HTML

Often called:

- Hash table
- Associative array
- (Self-balancing) binary search tree.



# What can we use for keys?

We do not want to be able to change the key once it is in the dictionary.

Data is accessed using a key instead of an index

Strings, numbers, tuples (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
{'value': 100.0, 1: 1, 2: 2, 2.5: 'hello', (): []}
>>> my_dictionary[()] = 'hello world'
>>> my_dictionary
{'value': 100.0, 1: 1, 2: 2, 2.5: 'hello', (): 'hello world'}
```

# Growing a dictionary

The dictionary, like the list, 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 delete key value pairs too

Use the del statement

```
>>> my_dict = { 1: 'first', 2: 'second', 3: 'third', 3.5: 'third and a halfthth'}
>>> my_dict
{1: 'first', 2: 'second', 3: 'third', 3.5: 'third and a halfthth'}
>>> del my_dict[3.5]
>>> my_dict
{1: 'first', 2: 'second', 3: 'third'}
```

# We can use a dictionary in many ways

We can use it to store:

- key value pairs that represent the attributes of some single object, like a character
- a single piece of information about many kinds of objects, like this.

```
exam_results = {
      'abraham' : 50.0,
      'betty' : 60.0,
      'chuck' : 70.0,
      'dolorys' : 80.0,
      'edgar' : 90.0,
      'ferne mae' : 99.9
}
```

# Iteration and Views

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

Remember dictionaries?

```
meals = {‘bfast’:’egg’, ‘lunch’:’poké’, ‘dinner’:’spinach’}
keys = meals.keys()
values = meals.values()
entries = meals.items()
```

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

## A virtual sequence (like the range object we use)

**Used for looping**

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

* https://docs.python.org/3/library/stdtypes.htm

# For Loops

```
>>> 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
```

# For Loops

Used with sequence types

Will Loop over every element in the given sequence or view

Very efficient

```
for x in sequence:
    do something
```

Like other languages we can **break** and **continue**

# For Loops

```
for name,age in [(“James”, 22), (“Danica”, 35), (“Rohit”,28)]:
     if age > 30:
           print(“Name: {0}, Age: {1}”.format(name, age))
           break


for val in "string":
    if val == "i":
        continue
    print(val)

    print("The end")
```

# While Loops

```
i = 1
while i < 6:
    print(i)
    i += 1
```

Use while loops when you need to loop over something that is not a sequence.

Remember to update the test expression!

# Treat Python With Care

# Python Pitfalls

- At first glance python feels more like a scripting language without rigid rules.
- It's "Loosey Goosey"
- It is tempting to use Python to write bad code
- We will come across situations where this flexible nature of python is an asset

# That's All For Week 1!

**Quizzes start next week!**

- The Quiz next week will be during the second lecture (to give you more time to prepare). This is **only** for next week!
- Next week we will cover Ranges and Slicing. This will be followed by Scope. That will give you all enough knowledge to get familiar with Python! (If you aren't already).
- Next week we will also get into OOP concepts and all the good stuff!