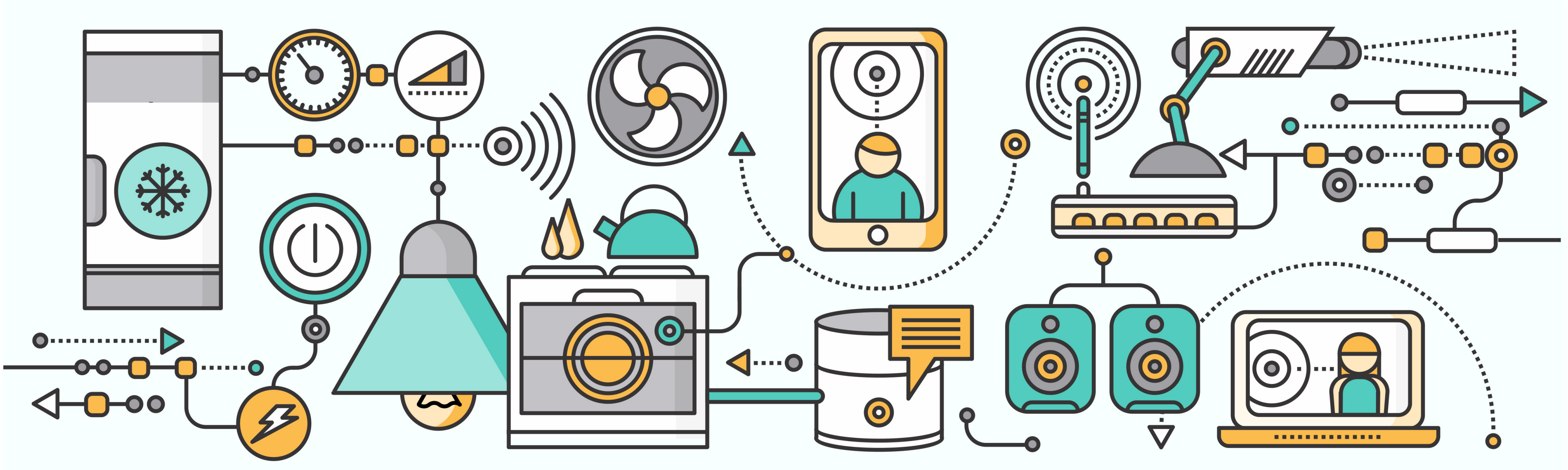
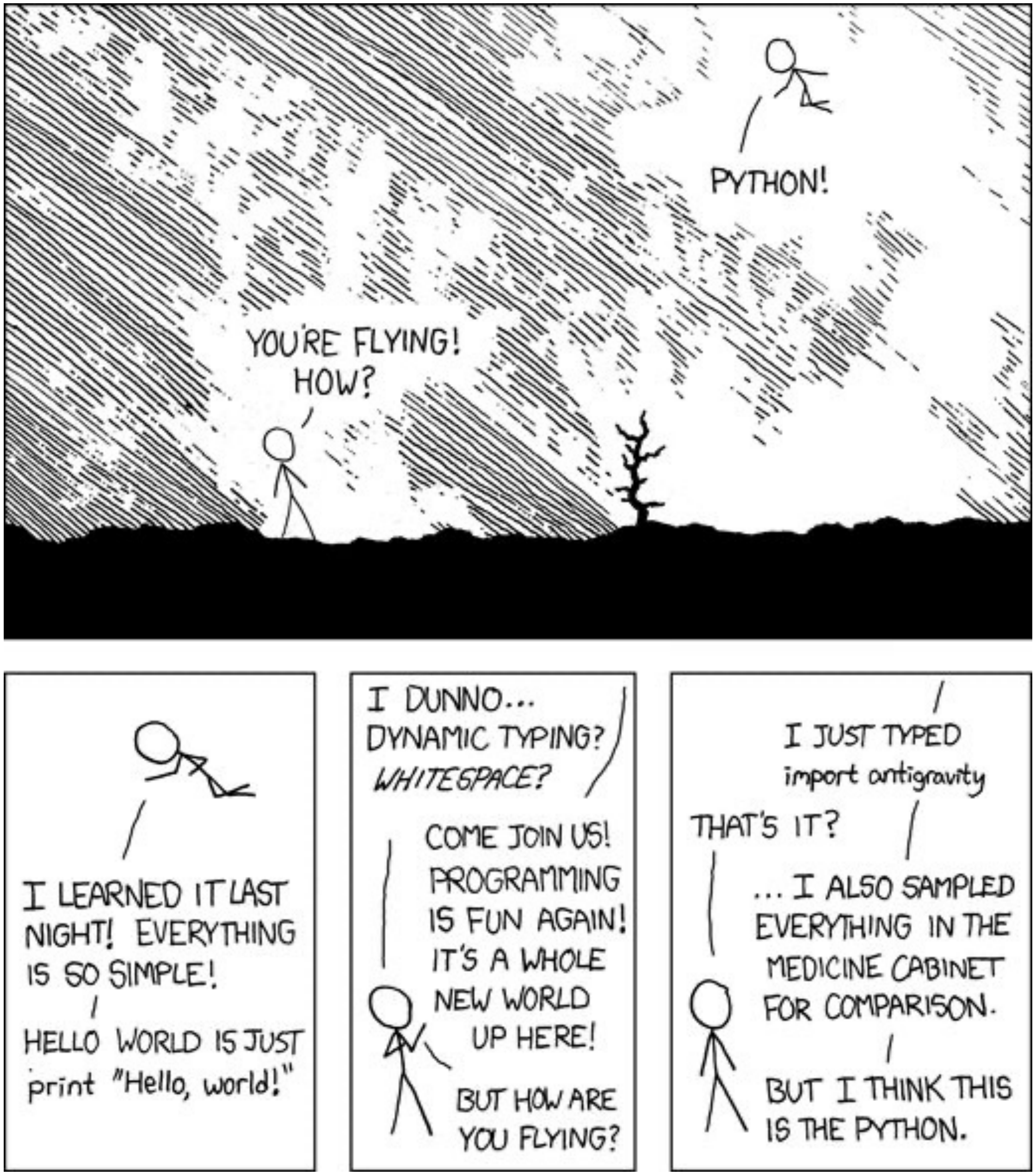
ENME 441
Mechatronics and the Internet of Things

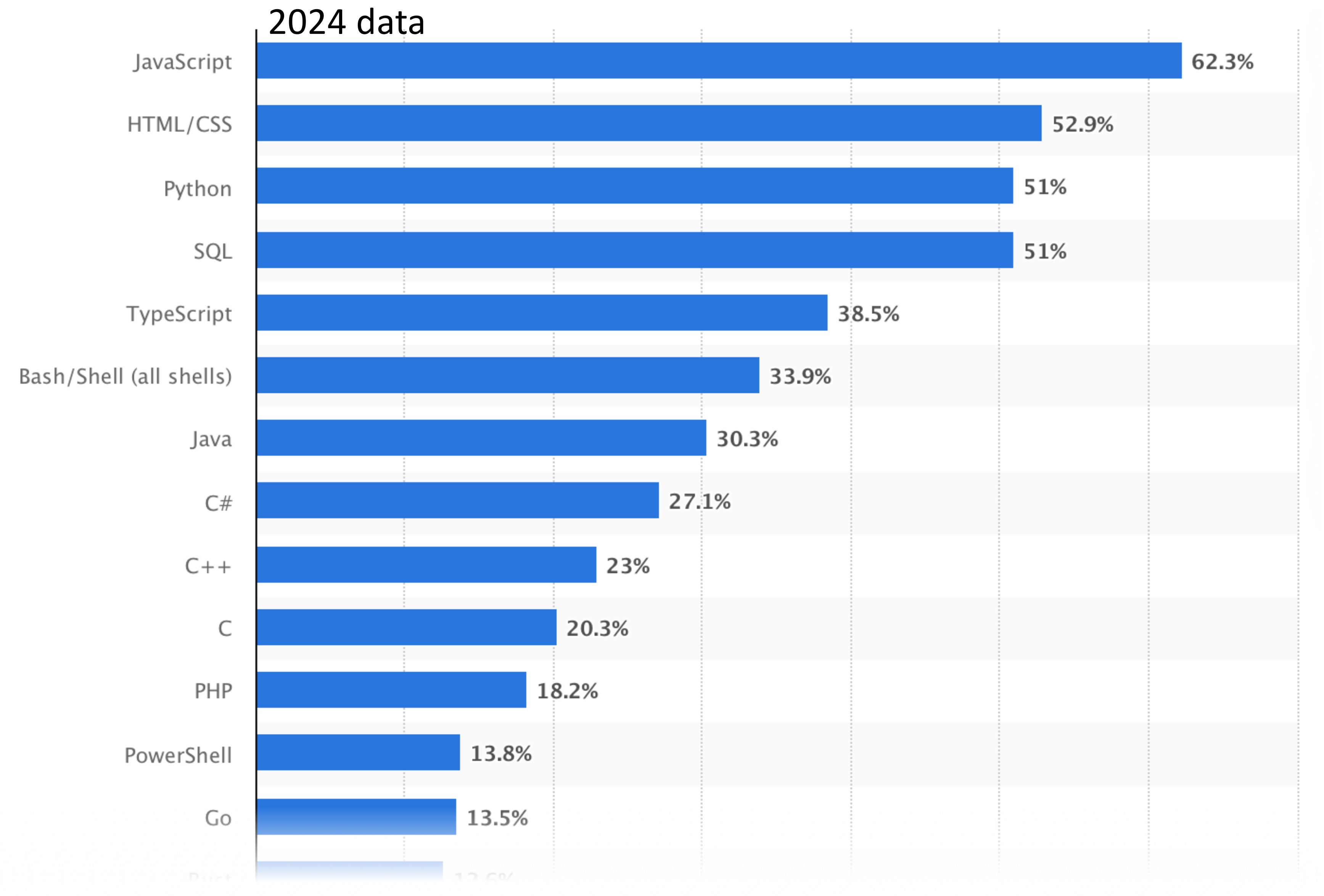


Python Programming

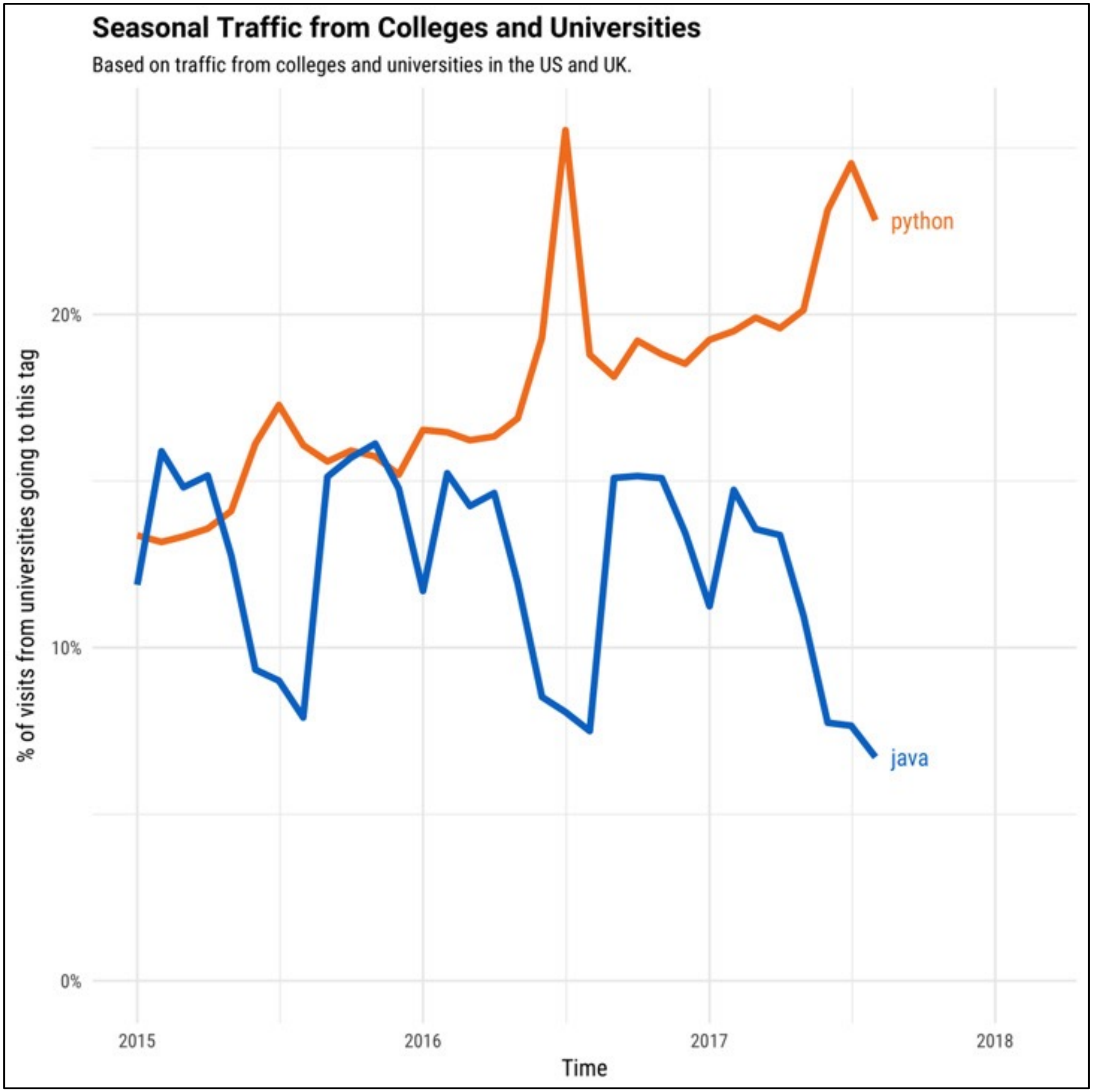
import antigravity


PYTHON!
YOU'RE FLYING! HOW?
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.

Python Programmers are In Demand


2024 data

https://www.statista.com/statistics/1296727/programming-languages-demanded-by-recruiters/

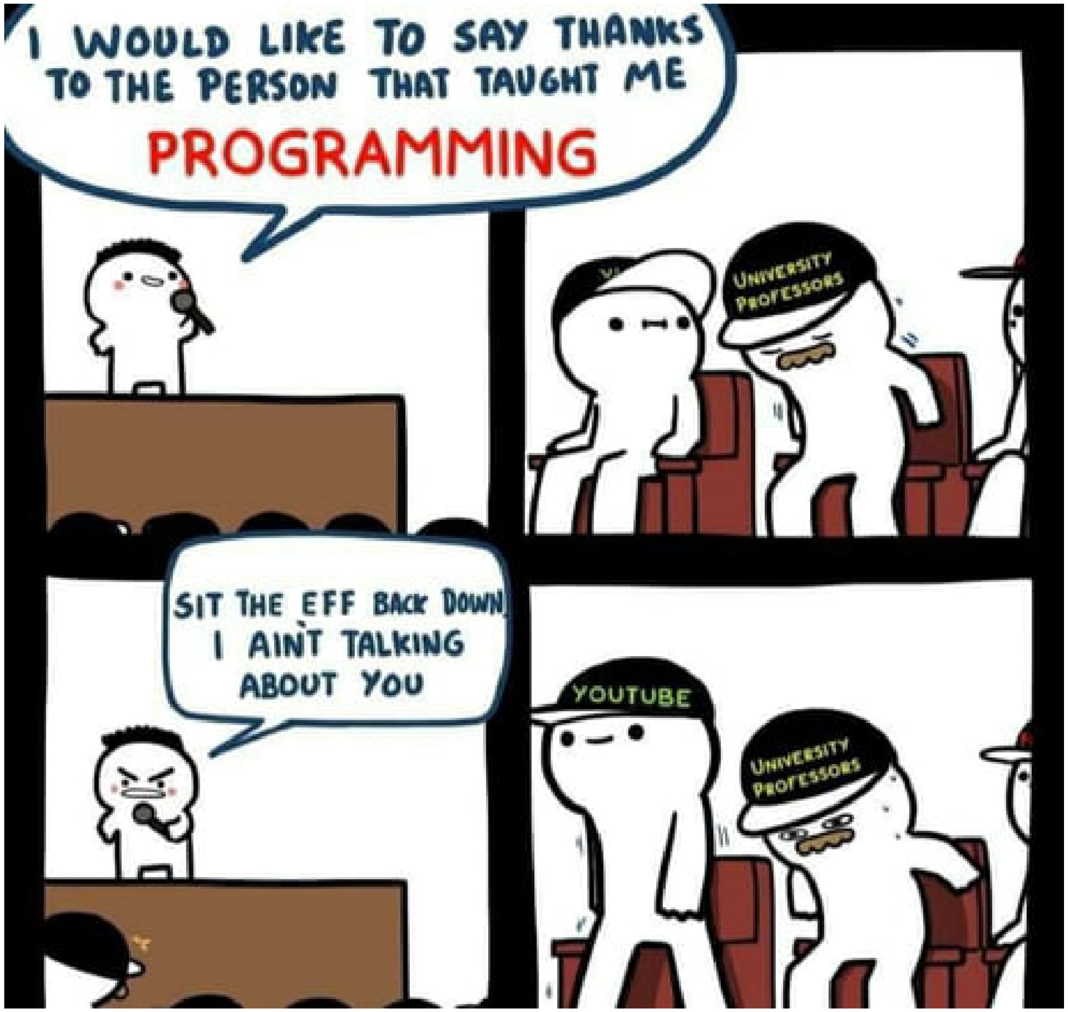


https://www.peerbits.com/blog/factors-will-drive-python-growth-in-2018.html

Virtual Python

- We will use http://repl.it ("replit") to learn Python
  - Create a personal Replit account and keep it bookmarked
  - Use Replit to quickly test code before working directly on your Pi Zero.
  - Your repls are public! anyone who knows your account name can access all of your code

- Many good online Python tutorials online via repl.it, Linkedin Learning and YouTube


I WOULD LIKE TO SAY THANKS TO THE PERSON THAT TAUGHT ME PROGRAMMING
UNIVERSITY PROFESSORS
SIT THE EFF BACK DOWN I AIN'T TALKING ABOUT YOU
YOUTUBE
UNIVERSITY PROFESSORS

Python Features

http://python.org

- Interpreted language (no compiling)
- Dynamically typed (no type declarations – type determined by context)
- Strongly typed (intermingling of variable types restricted)
- Easy to write & read
- Automatic memory management
- Powerful exception handling
- Everything is an object
- Highly portable / cross-platform

Code Structure

Python is easier to use than C++

```
// C++ hello world

#include <iostream>
using namespace std;

int main () {
  cout << "Hello world" << endl;
  return 0;
}
```

```
# Python hello world

print('Hello world')
```

But we need to learn some new syntax (and a few new programming concepts):

```
// C++ while loop

int x = 0;
while (x < 5) {
    x++;
    cout << x;
}
```

```
# Python while loop

x = 0
while (x < 5):
    x += 1
    print(x)
```

Python 2 vs. Python 3

	Python 2.x	**Python 3.x**
Division (/)	Floor div for intergers: `3/2 → 1`	Returns float: `3/2 → 1.5`
Print statements	Parentheses optional: `print "hello"`	Parentheses required: `print("hello")`
`input()`	`input()` returns value	`input()` returns string
`raw_input()`	`raw_input()` returns string	`raw_input()` does not exist

Basics

- Python supports 4 primary numerical data types:

```
size = 40               # integer
f1 = 5.6                # float
bigNum = -2215L         # long integer
z = 3+1j                # complex
```

  - *integer* same as *long* in C (at least 32 bits)
  - *float* same as *double* in C (at least 53 bits)
  - *long* has infinite precision
  - No double, signed, etc.
  - Complex number functions & methods:

```
abs(z), z.real, z.imag
```

- Strings:

```
'hello world', "bye world", 'it\'s hot', "that's nice"
```

Type Conversion

```
# Convert x to an integer:
int(x)

# Convert string s, written in given base, to an integer:
int(s, base)    # try: int('101',2) or int('A',16)

# Convert x to a long integer:
long(x)

# Convert x to a floating-point value:
float(x)

# Create a complex number from real & imaginary parts:
complex(real, imag)

# Convert x to a string:
str(x)
```

- *Python does not support C-style typecasting, e.g.* `(int)x`

Comments

```
# this is a single-line comment

"""
This is a multi-line
comment (placed
between triple full quotes)
"""
```

- Use comments to explain what is happening at different lines or blocks in your code

- Each custom function should be commented to explain what it does, what arguments it takes, and what value(s) it returns

- Be careful when using copy & paste with quotes to avoid special characters in your Python code!

Arithmetic Operators

Operator	Description
+	Addition
−	Subtraction
*	Multiplication
/	Division
//	Floor Division (e.g. 20 // 3 = 6)
**	Exponent (e.g. 2**3 = 8)
%	Modulo (remainder, e.g. 8 % 3 = 2)

Comparison Operators (Boolean output)

Operator	Description
==	equality
!=	inequality
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

Assignment Operators

Operator	Example	Same As
`=`	`x = 2`	`(basic assignment)`
`+=`	`x += 2`	`x = x+2`
`-=`	`x -= 2`	`x = x-2`
`*=`	`x *= 2`	`x = x*2`
`/=`	`x /= 2`	`x = x/2`
`//=`	`x //= 2`	`x = x//2`
`**=`	`x **= 2`	`x = x**2`
`%=`	`x %= 2`	`x = x%2`

String Operations

- Concatenate:

```
word = 'ab' + 'cd'
```

- Indexing & slicing:

```
'Hello'[0]    # = 'H'    <-- start counting at zero
'Hello'[1:3]  # = 'el'   (slice)
'Hello'[2:4]  # = 'll'   (slice)
'Hello'[-1]   # = 'o'    (last character)
```

- Finding string length:

```
len('abcde')  # works with any iterable, not just strings
```

- Strings are immutable: once created, they cannot be changed

String Methods

```
s.lower()
s.upper()
s.capitalize()
s.strip()
s.isalpha()
s.isdigit()
s.isspace()
s.startswith(string_to_check)
s.endswith(string_to_check)
s.count(string_to_count)
s.find(string_to_find)    # find first occurrence
s.rfind(string_to_find)   # find last occurrence
s.replace(old_string, new_string)
s.split(delimiter)
s.join(iterable)
```

(and many more!)

String Formatting

Given:

```
x = 2.5
y = 'half of'
z = 5
```

the following statements are equivalent:

```
s = str(x) + ' is ' + y + ' ' + str(z)
s = '%2.1f is %s %d' % (x,y,z)
s = '{:2.1f} is {:s} {:d}'.format(x,y,z)
```

f = float

s = string

d = decimal (integer)

b = binary

e = exponent

x/X = hexadecimal (lower/upper case)

See Python docs for many more formatting options:
https://docs.python.org/2/library/string.html

Basic printing:

```
print(str(x) + ' is ' + y + ' ' + str(z))
```

- no control over formatting

C style syntax:

```
print('%2.1f is %s %d' % (x,y,z))
```

- `%width.precision`

  `width` = <u>minimum</u> string length, with leading spaces added (if needed)

  `precision` = # digits after the radix point (decimal point)

String `format()` method:

```
print('{:2.1f} is {:s} {:d}'.format(x,y,z))
```

- more formatting options than C-style format control
- can simplify to `print('{:2.1f} is {} {}'.format(x,y,z)` since the string and integer values have fixed formats
- can use leading numbers to specify argument order:

```
 print('{2:2.1f} is {1:s} {0:d}'.format(z,y,x)
```

Formatted String Literals: f-strings

- Syntax: `f'text{expression}'`
- Values or expressions can be inserted at any location:

```
f'5 modulo 3 = {5%3}'

x = 2
y = 5
f'x + y = {x+y}'

x = 2.5
t = 12
f'{x} * cos({t}) = {x*math.cos(t)}'
```

- Add string formatting:

```
f'5 / 3 = {10/3:5.3f}'
```

Flow Control: `if`

- Syntax:

```
if Boolean_condition:
    code_block_when_True
```

- Example:

```
x = float(input("Enter a number greater than 5:"))
if x <= 5:
    print(f"{x:4.2f} is too small.")
    print("Maybe pay more attention next time?")
if x > 5:
    print(f"Your number is {x:4.2f}.")
```

- Note:
  - A colon defines the start of each `if` statement code block
  - Indentation defines the extent of each code block

Flow Control: `if...elif...else`

```
x = int(input("Enter #:"))
if x < 0:
    x = 0
    print("Negative value entered, changed to zero")
elif x == 0:
    print("Zero")
elif x == 1:
    print("One")
else:
    print("More than one")
```

- No *switch/case* statement in Python

Boolean Operators: `and, or, not`

- Boolean operators are used to compare logical values (`True` (1) or `False` (0))

- Boolean operators in Python are written in plain English:

  `and, or, not`

  ```
  if not ((x > 20 and y < 50) or z < 0):
      # do something
  ```

- Different from bitwise operators (`&, |, ~`) which will be discussed later

- Operator precedence:
  bitwise → numerical comparison → Boolean

Flow Control: `for`

- Syntax:

```
for loop_variable in iterable:
  # Do something.
  # Never modify the iterable within the loop!
```

- An iterable is a data structure that can be "iterated over". We will talk more about iterables soon, but for now consider a common iterable in Python, namely the `list` data structure:

```
my_list = ['cat', 'window', 'defenestrate']
for x in my_list:
    print(x, len(x))
```

- Python iterates through each element of the list, assigning the current list value to the loop variable (`x` in the above example) before executing the loop code.

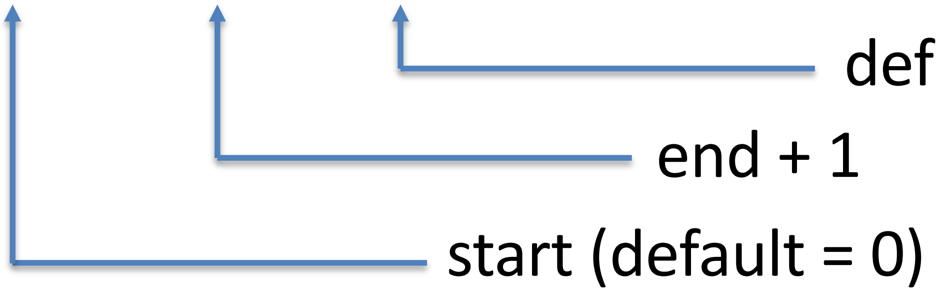
- Iterables can return their members one at a time:

```
for item in [1, 2, 3]:            # list
    print(item)

for item in range(2,1000):        # range object
    print(item)

for item in (1, 2, 3):            # tuple(immutable list)
    print(item)

for key in {'one':1, 'two':2}:    # dictionary (dict)
    print(key)

for char in "123":                # string (str)
    print(char)

for line in open("myfile.txt"):   # file (TextIOWrapper)
    print(line, end='')
```

Flow Control: `for` loops and `range()`

- The `range()` function returns an iterable of <u>integers</u>:

```
range(        stop)
range(start, stop)
range(start, stop, step)
```



default = 1

end + 1

start (default = 0)

- Examples using a `for` loop:

```
for i in range(5):          # [0, 1, 2, 3, 4]
    print(i)

for x in range(2, 15, 3):   # [2, 5, 8, 11, 14]
    print(x)
```

for loops: break, continue

```
for x in range(2,6):        # outer loop, x=2,3,4,5

    for y in range(1,6):    # nested loop, y=1,2,3,4,5

        if x == y:
            continue  # go directly to next y value

        # keep going if 'continue' not called...

        print(f'{x}, {y}')

        if x + y > 5:
            break  # exit inner loop, go to next x value
```

Looping with Index Values: `enumerate()`

- Consider a loop where we want to print the index value for each element of a list that is greater than 5. In C++ we might write something like this:

```
for (int i=0; i<theList.size(); i++) {
  if (theList[i] > 5) {
    cout << i;        # Print the loop variable
  }
}
```

- In Python, the loop might look like this:

```
theList = [4, -2, 98, 2.5, 6.02]
for value in theList:
  if value > 5:
    print(???)     # No loop variable to print!!
```

- To access the index values, use `enumerate()` to create an *iterable of tuples* consisting of (index, value) pairs:

```
for (i,value) in enumerate(theList):
  if value > 5:
    print(i)
```

Flow Control: `while`

```
a = -10
while a <= 10:
    print(a)
    a += 1    # same as a=a+1
```

```
import time
while True:  # infinite loop
    print('.')
    time.sleep(0.25)   # pause for 1/4 second
```

```
while True:
    if int(input('enter number > 10 to end: ')) > 10:
        break
```

Do Nothing: `pass`

- `pass` does nothing – use it as a “syntactic filler”:

```
while 1:
    pass
```

Assignments Inside Loop Expressions

- Unlike C/C++, assignments *cannot* occur inside a loop expression:

✗

```
while (x = f.readline()) != "end":
    # do something
```

✓

```
x = f.readline()
while x != "end":
    x = f.readline()
    # do something
```

Defining Custom Functions

- Function syntax:

```
def function_name():
    # function code goes here
    # and ends when indents stop
```

- Example:

```
# Print a Fibonacci series up to n, and
# display the result:
def fib(n):
    a, b = 0, 1
    while b < n:
        print(b)
        a, b = b, a+b
    print(a)
```

- Functions must be defined ahead of the first point in the code where they are called.

Passing & Returning Values

```
# find maximum of two values:
def maximum(x,y):   # Pass multiple parameters
    if x>y:
        max = x
    else:
        max = y
    return max      # Return the result



# call the function:

print(maximum(2,3))       # order matters

print(maximum(y=3, x=2))  # order doesn't matter
```

- A more compact version:

```
def maximum(x,y):
    if x>y:
        return(x)
    else:
        return(y)
```

- Even more compact:

```
def maximum(x,y):
    if x>y: return(x)
    else: return(y)
```

Recursive Functions

- Functions can call themselves (recursion):

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Default Argument Values

- Default parameters allow functions to be called with only a subset of the arguments:

```
def maximum(x, y=5):
    if x>y: return(x)
    else: return(y)


print(maximum(1))   # returns 5

print(maximum(6))   # returns 6

print(maximum(1,2)) # returns 2
```

- Arguments with default values must occur at the end of the parameter list

Arbitrary Number of Arguments

- A function can accept a variable number of arguments by adding * before the argument name:

```
def my_sum(*vals):
    sum = 0
    for v in vals:
        sum += v
    return sum
```

- Can combine with standard arguments:

```
def sum_prod(m, *vals):
    sum = 0
    for v in vals:
        sum += v
    return m*sum
```

Iterators via Generators

- Iterators are iterable objects that can be manually iterated over
- Generators are functions that can be used as iterators
- `next(f)` tells function `f` to run until reaching `yield`
- `yield` returns a value for the current iteration
- The function state is retained following each `next()` call

```
# Define a generator function:
def fibonacci():
    n1=0
    n2=1
    while True:
        yield n1     # value to return for each iteration
        n1, n2 = n2, n1 + n2

f = fibonacci() # declare an iterator

# Print the 1st 10 values in the Fibonacci sequence:
for _ in range(10):
    print(next(f))
```

- A generator can contain multiple yield statements (instead of a single statement in a loop as it the previous example):

```
def gen(x):
    yield x**2
    yield x**3
    yield x**4

x = gen() # declare  a generator

for _ in range(3):
    print(next(x))
```

- Calling `next()` after iterating through all generator values will yield an error.
- Generators cannot be “reset”, but a new generator can be defined using the same function.

Iterators via `iter()`

- The `iter()` function takes an iterable object as a function, and returns an iterator
- This allows us to manually iterate over the values in the original iterable:

```python
mylist = [5,1,0,6,7]

it = iter(mylist) # declare an iterator

print("Here is the first value in the list:")
print(next(it))

print("Now the second value:")
print(next(it))

print("And the sum of the rest:")
print(next(it) + next(it) + next(it))
```

Global vs. Local Variables

- Global variable values can be accessed within any function.

```
c = 5      # global variable
def add1(x):
    return(c + x)

c = add1(3)
```

- Use `global` keyword to modify a global variable inside a function:

```
c = 5      # global variable
def add2(x):
    global c
    c += x

add2(3)    # no need for assignment
```

Lists, Tuples, Sets, and Dictionaries

Beyond strings, 4 common iterable data structures supported by Python are:

List

mutable & ordered data sequence

Tuple

immutable & ordered data sequence

Set

mutable & unordered collection with no duplicates

Dictionary

mutable & unordered sequence of (key, value) pairs

All of these structures can hold heterogeneous data types

Lists

```
# Lists are mutable (elements can be changed) and
# ordered (position in the sequence is fixed):
a = ['foo', 'bar', 100, 123.4, 2*2]

# Indexed and slicing:
a[0]     # result is 'foo'
a[1:2]   # result is ['bar']
a[1:3]   # result is ['bar', 100]
a[:2]    # result is ['foo', 'bar']
a[2:]    # result is [100, 123.4, 4]

# Lists can contain lists (or any other Python data structure):
myList = [1, 2, [3, 4, 5]]    # 3rd element is a list
myList[2][1]                  # result is 4

# Assignments:
a[2] += 23      # change element value
a[0:2] = [1,12] # reassign multiple elements
a[0:2] = []     # delete elements 0 and 1 (or use del a[0:2])

# Finding list length:
len(a)          # result is 3 after del operation above

# Create an empty list of n elements:
a = n*[None]
```

List Methods

```
l.append(x)      # append x to end of list

l.extend(L)      # append all items in list L

l.insert(i,x)    # insert x at index i

l.remove(x)      # remove the first item in list with value x

l.pop(i)         # remove and return ith value

l.pop()          # remove and return last value

l.index(x)       # return index for 1st value x

l.count(x)       # return number of times x appears in list

l.reverse()      # reverse the list

l.sort(key=None) # sort items (with or w/o key function)
```

Removing List Items: del, pop, remove

```
a = ['foo', 'bar', 100, 1234, 2*2, 5, 'foobar', -1]
```

- `del`

```
del a[0]    # Remove 1st element, same as a[0:1] = []
del a[2:4]  # Remove slice 2 & 3, same as a[2:4] = []
```

- `pop()`

```
val = a.pop()    # Remove last element from the list,
                 # and assign it to val
val = a.pop(2)   # Remove the 3rd element,
                 # and assign it to val
```

- `remove(value)`

```
a.remove('bar')   # Remove first element of the list
                  # with the given value
```

Checking Membership: `in`

- We have seen how `in` can be used to iterate through an iterable object:

```
for i in range(10):
    print(i)
```

- The `in` keyword can also be used to test membership in any iterable:

```
the_list = ["this", "that", "those", "these", "them"]
if "this" in the_list:
    print("the list contains 'this'")
```

```
the_string = 'abracadabra'
if 'c' in the_string:
    print(f"c is at position {the_string.find('c')}")
```

List Sorting with a Key Function

```
def myFunc(e):
    return len(e)


colors = ['blue', 'red', 'green', 'orange']

colors.sort(key=myFunc)

print(colors)
```

- This code can be simplified by using a *lambda function*…

Lambda Functions

- A lambda function is a small single-expression function
- Lambda functions can be anonymous (unnamed) when defined in-line with other code:

```
lambda bound_variables: function_body
```

Regular function:

```
def add(x,y):
   return x+y

sum = add(1,2)
```

Equivalent Lambda function:

```
add = lambda x,y: x+y


sum = add(1,2)
```

Lambda Functions

- Anonymous lambda functions can be used to expand the capabilities of other functions:

(1) Lambda functions in return statements:

```
def func(x):
    return lambda a: x**a

pow2 = func(2)    # new function: pow2(n) = 2**n
pow10 = func(10)  # new function: pow10(n) = 10**n
```

(2) Lambda functions as arguments:

```
colors = ['blue', 'red', 'green', 'orange']
colors.sort(key=lambda x: len(x))

vals = [2, -3, -4, 6, 3]
vals.sort(key=lambda x: x*x)
```

List Comprehension

- List comprehension is a compact syntax to create a list from an iterable sequence
- Significantly faster than using a standard loop

```
[i for i in iterable]
```

- Example:

```
# Standard for loop:
x = [1, 3, 5, 2]
y = []
for val in x:
    y.append(val**3)

# List comprehension:
x = [1, 3, 5, 2]
y = [val**3 for val in x]
```

List Comprehension: Embedded `for` Loops

```
v1 = [2, 4, 8]
v2 = [4, 3, -9]
```

- Embed a `for` loop during list construction:

```
w = [val**val for val in v1]
```

- Cross product:

```
[x*y for x in v1 for y in v2]

[x+y for x in v1 for y in v2]
```

- Dot product:

```
[v1[i]*v2[i] for i in range(len(v1))]
```

List Comprehension: Embedded Conditionals

```
v1 = [2, 4, 8]
v2 = [4, 3, -9]

z1 = [3*x for x in v1 if x > 3]

    x = 2 !> 3                 →      z1 = []
    x = 4 > 3    →  3*4 = 12  →      z1 = [12]
    x = 8 > 3    →  3*8 = 24  →      z1 = [12  24]

z2 = [x*y for x in v1 for y in v2 if x*y > 0]

    x*y = 2*4 = 8 > 0   →      z2 = [8]
    x*y = 2*3 = 6 > 0   →      z2 = [8 6]
    x*y = 2*-9 !> 0     →      z2 = [8 6]
    x*y = 4*4 = 16 > 0  →      z2 = [8 6 16]
    x*y = 4*3 = 12 > 0  →      z2 = [8 6 16 12]
    x*y = 4*-9 !> 0     →      z2 = [8 6 16 12]
    x*y = 8*4 = 32 > 0  →      z2 = [8 6 16 12 32]
    x*y = 8*3 = 24 > 0  →      z2 = [8 6 16 12 32 24]
    x*y = 8*-9 !> 0     →      z2 = [8 6 16 12 32 24]
```

Copying Lists: Deep vs. Shallow Copies

- The assignment operator (=) does not copy lists – it simply creates a “reference binding” between the target and new object names:

```
a = [2, 4, 6]
b = a
print(a)     # yields [2, 4, 6]
b[1] = -1
print(a)     # result is [2, -1, 6] (changed!)
```

- Thus `b` is a “shallow copy” of `a`, since both names point to the same memory locations (and thus the same list values):

```
id(a) == id(b)        # yields True
```

Copying Lists: Deep vs. Shallow Copies

- Create a deep (true) copy by slicing the original array:

```
a = [2, 4, 6]
b = a[:]         # slice the original list for deep copy
print(a)         # yields [2, 4, 6]
b[1] = -1
print(a)         # result is [2, 4, 6] (a unchanged)
id(a) == id(b) # False
```

- However, if the original list contains elements that are themselves lists (or other nested objects), those elements will be referenced, **not** copied – thus slicing does not create a deep copy for a nested list!
- When working with nested lists, use `deepcopy()` for a full deep copy:

```
from copy import deepcopy
b = deepcopy(a)
```

Tuples

- Tuples are similar to lists but are immutable (as a result they are faster than lists)

- Define using values separated by commas:

```
t = (123, 543, 'bar')  # parentheses not required

print(t)    # = (123, 543, 'bar')

print(t[0]) # = 123
```

- Tuples may be nested:

```
u = t, (1,2)

print(u)      # = ((123, 542, 'bar'), (1,2))
```

Tuple Methods

`t.count()`	returns occurrences of element in a tuple
`t.index()`	returns smallest index of element in tuple

Other Iterable Functions

`any()`	checks if any element of an iterable is True
`all()`	returns true when all elements in iterable are true
`len()`	returns length of an object
`max()`	returns largest element
`min()`	returns smallest element
`map()`	applies a function and returns a List
`reversed()`	returns reversed iterator of a sequence
`slice()`	creates a slice object specified by range()
`sorted()`	returns sorted list from a given iterable
`sum()`	add items of an iterable
`tuple()`	creates a Tuple
`zip()`	returns an Iterator of paired Tuples
`enumerate()`	returns an Enumerate object (same as zip(), but with sequential key values)
`filter()`	constructs iterator from elements which are true
`iter()`	returns iterator for an object

Sets

- A set is an unordered collection of unique elements.
  - Useful for membership testing & eliminating duplicate values from a list.
  - Set objects support mathematical operations like union, intersection, and difference.

- Curly braces `{}` or the `set()` function can be used to create sets:

```
a = set('abracadabra')    # {'a','r','b','c','d'}

b = {5, 10-9, 10<9}
```

- To create an empty set use `set()`, not `{}` (which creates a dictionary)

Set/List Conversion

- A set can be converted to a list (and visa versa).
- This makes it easy to remove duplicate elements from a list:

```
a = 'abracadabra'    # A string is just a list of chars
a = set(a)           # {'a','r','b','c','d'}
a = list(a)          # ['a','r','b','c','d']
```

Dictionaries

- A Python *dictionary* is a set of *key*:*value* pairs

  key = string or integer

  value = any data type (including another dictionary)

- Keys must be unique
- Order does not matter
- Elements can be easily accessed, deleted, or replaced based on their key

Dictionary Example

```
foo = {}                   # empty dictionary
foo["word"] = "hi there"   # add string
foo["count"] = 99          # add integer
foo["list"] = ["bar", 25]  # add list
foo[4] = "foobar"          # use integer for key

print(foo)
print(foo["word"])
print(foo["list"][1])
print(foo[4])
```

Manipulating Dictionaries

```
del foo["count"]          # delete the given entry
foo["word"] = 26          # changes value (and type!)
foo["new word"] = "yo"    # add new entry
```

- Dictionaries can also be constructed directly without starting with a null dictionary:

```
foo2 = {"foobar":123, "this":"that", 2:(123, 'x')}
```

- Just as with lists, the assignment operator (=) does not copy dictionaries – use `copy.deepcopy()` instead

Dictionary Methods

`d.keys()` Return dictionary keys as a *dict_keys* object containing a list of keys

`d.values()` Return dictionary values as a *dict_values* object containing a list of values

`d.items()` Return dictionary key:value pairs as a *dict_items* object (iterator of tuples)

- Use `list()` to convert keys, values, or items (key:value pairs) to a list, allowing all list methods to be applied to dictionary data

Iterating on a Dictionary

- Dictionaries key and value can be retrieved at the same time using the `items()` method, which returns an *iterator of tuples*:

```
knights = {
  'gallahad': 'the pure',
  'robin': 'the brave' }
for (key, value) in knights.items():
  print(key, value)
```

- Similar behavior can be achieved by looping through two lists simultaneously with paired values using `zip()`, which also returns an *iterator of tuples*:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'to seek the holy grail', 'blue']

for q,a in zip(questions, answers):
   print(f'What is your {q}? It is {a}.')
```

Exception Handling: `try`, `except`

- When an error occurs during execution, the interpreter “throws an exception” and the code stops.
- We can “catch” these exceptions and decide what the code should do, instead of simply stopping.
- Here is a simple example:

```
while 1:
  try:
    x = int(input("Enter a number: "))
  except ValueError:
    print("Not a valid number")
  # code continues here, even if there was an exception
```

- First, execute the `try` block
- If no exception, skip the `except` block (and continue)
- If an exception occurs, execute the `except` block (and continue)

Exception Handling: `else`, `finally`

- A more complicated example:

```
import math
while 1:
  try:
    x = input("Enter a number >= 0: ")
    y = math.sqrt(float(x))
  except ValueError:   # will catch both <0 and strings
    print("Not a valid number")
  else:    # execute only if no exception
    print("Still doing stuff")
  finally: # execute regardless of exception
    print("Go again")
```

- Execute the `else` block only if there is no exception in the `try` block.
- Execute the `finally` block always, regardless of whether an exception is thrown in either the `try` block or the `except` block, and even if the `try` block terminates (for example by returning from a function call)

Exception Handling Example

- Example using file I/O:

```
filename = 'data.txt'

try:
    f = open(filename, 'r')

except IOError:
    print(f'cannot open {filename}')

else:
    for (linenum, linetext) in enumerate(f):
        if 'my text' in linetext:
            print(f'found in line {linenum}')
    f.close()

finally:
    print('All done')
```

Displaying Exception Details

```
try:
    print(1/0)

except Exception as e:
    print(e)
    print(e.__doc__)
```

- Displaying exception details can help with debugging.

- No exception type defined – most exceptions are members of the `Exception` class and will be caught, but some exceptions will get through

- You can use a simple “`except:`” statement to catch ALL exceptions, but note that this includes `KeyboardInterrupt` (ctrl-C) which may cause problems when trying to exit your code

Built-In Exceptions

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

Some built-in exceptions:

- `ZeroDivisionError`
- `FloatingPointError`
- `EOFError`
- `KeyboardInterrupt`
- `FileExistsError`
- `FileNotFoundError`

Custom exceptions can be created by extending `Exception` or `BaseException` class

Importing From Modules (Libraries)

```
import theModule
```

or

```
from theModule import theFunction
```

For example, the following code blocks yield identical results:

```
import random
random.randint(5,20)

from random import randint
randint(5,20)
```

Python Standard Library

Common modules from the standard library:

```
time (time.sleep, time.time)
datetime
sys
os.path (os.path.basename, os.path.dirname)
math (math.exp,  math.log, math.sin, etc.)
cmath (complex variable math)
random
json
cgi
urllib2
```

Custom Modules

- Create a normal python file containing desired functions, e.g. `new_module.py`
- Place it in the same directory as your main code
- Add an import statement at the top of the main code:

```
import new_module
```

  - *Note there is no ".py" at the end of the import statement, even though the file has the extension.*

if __name__ == "__main__"

- __name__ is a built-in variable that evaluates to "__main__" if the current module is executed directly (rather than being imported).
- We can prevent a given code block from executing upon import as follows:

```
def your_custom_function():
  # put main Python code here, with the function
  # definition at the top of the code file

if __name__ == "__main__"
  your_custom_function()
```

- Example:

Contents of foo.py

```
def main():
  print("foo main() executed")
if __name__ == '__main__':
  main()
else:
  print("foo imported")
```

Contents of bar.py

```
import foo

# try toggling this on/off:
foo.main()
```

Breaking a Long Line

- The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces.
  - If necessary, add an extra pair of parentheses around an expression
- Alternately, add a backslash at the line end to denote continuation on the next line.
- Make sure to indent the continued line appropriately.

break after open parenthesis or commas

```
params = urllib.urlencode(
   {'field1: random.randrange(100,1000),
    'field2: random.randrange(100,1000),
    'key':"74W7TQ5E82VFO8GG"} )
```

indent additional lines

File I/O

- To access a file, a file object must be created:

```
f = open("myfile.txt")
```

- Here `f` is the file object that can access the *methods* of the *class* to which they belong, including:

```
line = f.read() # read file (from current position
                # to the end) into a string
                # including newline characters

f.write('new text')  # adds string to file at
                     # current position

f.close()     # close the file

f.seek(n)     # set file position
              # n = byte value (1 byte per character)
              # n = 0 for start of file
```

File I/O

- Common file opening modes:

`f = open(filename, 'r')`	← read (default, can omit ‘r’) (file must exist)
`f = open(filename, 'w')`	← write by overwriting file (create new if none)
`f = open(filename, 'a')`	← write by appending to end (create new if none)
`f = open(filename, 'x')`	← write new file only (error if existing file)
`f = open(filename, 'r+')`	← read + write (file must exist, content preserved)
`f = open(filename, 'w+')`	← read + write (overwrite, create new if none)

- Text encoding is platform-dependent (e.g. cp1252 in Windows, utf-8 for just about everything else) – specifying encoding can make your code more robust:

```
f = open("myfile.txt", mode = 'r', encoding = 'utf-8')
```

- Files should always be closed after read or write operations:

```
f.close()
```

File I/O – File Objects as Iterators

- A file object can be thought of as a *list of strings*, with each line of the file treated as a separate element in the list.
- Each line's newline character is included in the string.
- File objects are iterators, with each iteration yielding the next line in the file:

```
f = open("data.txt", 'r')
for line in f:
  print(line.strip())    # strip() removes newline
f.close()
```

- File objects can *only be traversed once*.
  - To traverse a file multiple times, close then create a new file object
  - Can also use `seek()` to reset the file pointer to the start:

```
f.seek(0)
```

Accessing Files using `with`

- Python’s `with` statement can be used to simplify the syntax for handling errors with file I/O:

```
with open("test.txt", mode = 'r+') as f:
  f.write("hello there")
  # f.close() not needed
```

- The file will automatically close at end of the `with` block (even if an exception is raised).

`with open` *statements are not (yet) supported in MicroPython*

Example 1

Write Python code to:

1. Query the user for a number
2. Compare the given number to all values within a file called `data.txt`
3. Display all numbers in the file that are larger than the user-defined number.

You may assume that `data.txt` exists, and that the file holds a single real number on each line.

Try it using repl.it:

- Create a new Python repl
- Create a new file `data.txt` containing a single number on each line
- Write & test your code

Example solution

```
m = float( input("Enter a number: ") )

with open("test.txt") as f:   # mode = 'r' is default
  for line in f:
    if float(line) > m:
      print(float(line))
```

- Note the we don’t need to `strip()` the line since `float()` will ignoring the newline during conversion

Example 2

A file `data.txt` contains a list of numbers, with one value per line. Write Python code to calculate the sum of all values in the file.

```
total = 0
with open("data.txt") as f:
    for val in f:
        total += float(val)
```

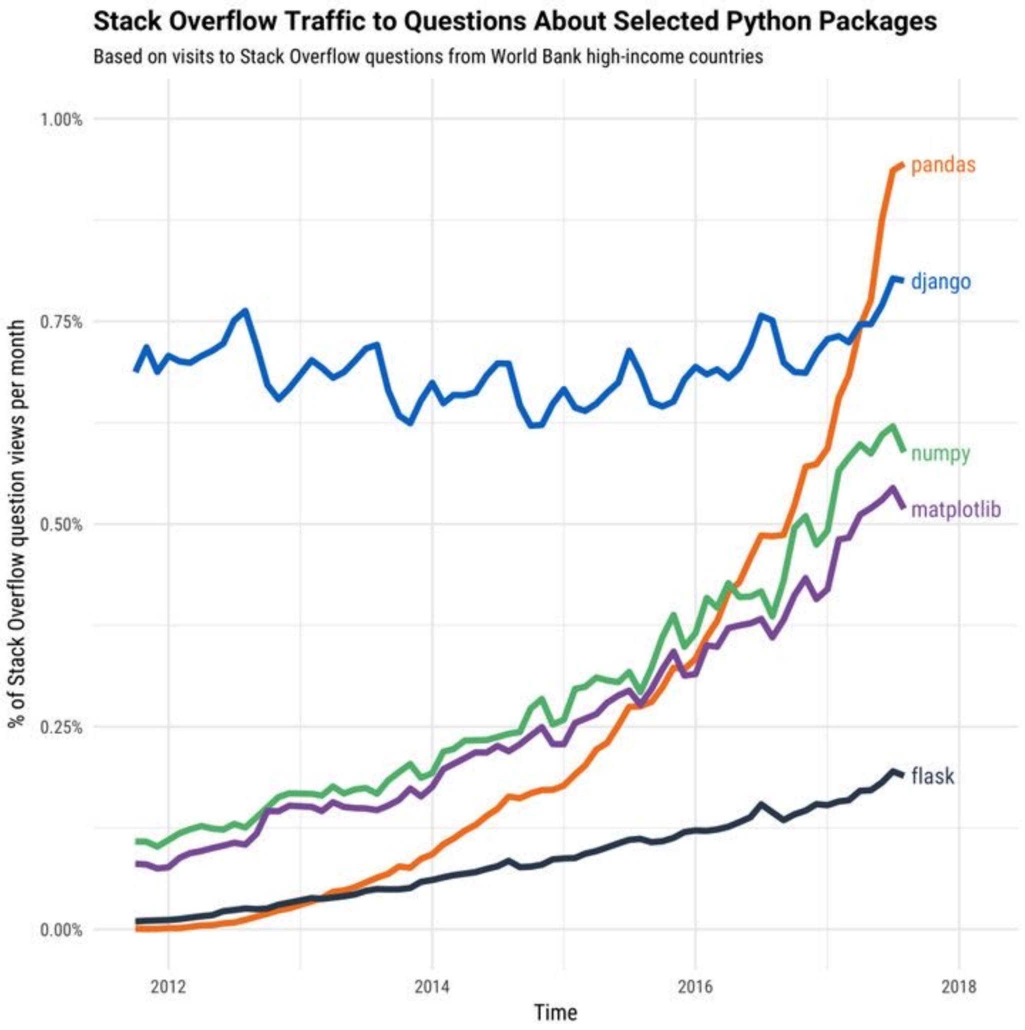
- Here is a more “Pythonic” solution:

```
with open("data.txt") as f:
    data = f.read().split('\n’)
    total = sum(float(n) for n in data)
```

- `read()` returns a string containing the entire file contents
- `split()` creates a list with elements determined by the given separator (newline)
- Use list comprehension to convert since `float()` cannot take a list as an argument

Important External Packages

- **numpy:** high-level math functions, multi-dimensional matrix operations, and AI/ML algorithms
- **pandas:** data manipulation and analysis (data structures and operations for manipulating numerical tables and time series)
- **matplotlib:** 2D plotting library, serves as the core of other high-level plotting packages
- **django, flask:** high-level frameworks for Web coding



*None of these packages are available for MicroPython. However, MicroPython has the built-in **ulab** module that provides functionality similar to **numpy***

What do we know at this point?

- Variables, scope, type conversion
- Strings, string methods
- Boolean operators
- Flow control:
  - `if` / `elif` / `else` statements
  - `for` loops
    - `range()` function
  - `while` loops
    - `break`, `continue`
- Custom functions
- Lists, tuples, sets, dictionaries (& their methods)
- Iteration and enumeration
- Exception handling
- Importing modules
- Keyboard input
- File I/O

Number Systems

The I^2C address was displayed as a hexadecimal (base 16) number using the `hex()` function. We can can also use string formatting to display values in different number systems:

`b` → binary (base 2)
`o` → octal (base 8)
`d` → decimal (base 10)
`x` → hexadecimal (base 16)
`X` → hexadecimal (base 16) upper case

```
print(f'{0x40:02d}')    # display hex as decimal
print(f'{0x40:08b}')    # display hex as binary
print(f'{0b1111:02X}')  # display binary as hex (upper case)
print(f'{0o31:02d')     # display octal as decimal
```

Number System Conversion

```
print(f'{0x40:02d}')    # display hex (base-16) as decimal
print(f'{0x40:08b}')    # display hex (base-16) as binary
print(f'{0b1111:02X}')  # display binary (base-2) as hexadecimal
print(f'{0o77:02d')     # display octal (base-8) as decimal
```

Decimal	Binary	Octal	Hexadecimal
1	00000001	001	01
2	00000010	002	02
4	00000100	004	04
8	00001000	010	08
16	00010000	020	10
32	00100000	040	20
64	01000000	100	40
128	10000000	200	80

Bitwise Operators

- See https://wiki.python.org/moin/BitwiseOperators

- Bitwise operations allow us to efficiently manipulate bit values within binary words, e.g.:

– **Bit**	single binary value (0,1)
– **Nibble**	4-bit word
– **Byte**	8-bit word

- No standard or maximum binary word length in Python:
  - Binary words can be infinite length

- Why do we care?
  - bit fields, bit masks, bit flags, finite state machines, graphics operations, compression, encryption, communications, etc.

Bitwise Operators

Bitwise operators:

```
&    # AND (ampersand)
|    # OR (bar)
^    # XOR (caret)
~    # NOT / one's Complement (tilde)
<< n # shift left by n bits
>> n # shift right by n bits
```

Combine bitwise operators with assignment operator:

```
x = 1       # binary: 0001
x <<= 2     # binary: 0100
x |= 3      # binary: 0111 (3$_{10}$ = 0011$_{2}$)
```

Base conversion using `int()` and `bin()`:

```
bin(5)             # convert b10 integer to b2 value (0b101)
x = int('0101',2)  # convert 4-bit b2 string to b10 (5)
                   # same as x = 0b0101
```

Bitwise lecture code: https://replit.com/@ddevoe/bitwise

Code Development Process

1. Think through the problem statement, and conceptualize the high level steps that need to take place.
2. Write each step as a comment in a new code file.
3. Look over your steps to see if something is being repeated – if so, think about making a separate function for that action.
4. Start filling in the code corresponding to each comment. When you don’t remember syntax, Google is your friend.
5. For loops, re-think your comments & code – English does not always reflect the real steps needed for flow control.
6. Run the code periodically to check functionality of individual pieces, and keep iterating from step 3 until the code works as desired.
7. Once the code works, look everything over again for places you can simplify and modularize before deciding you are done

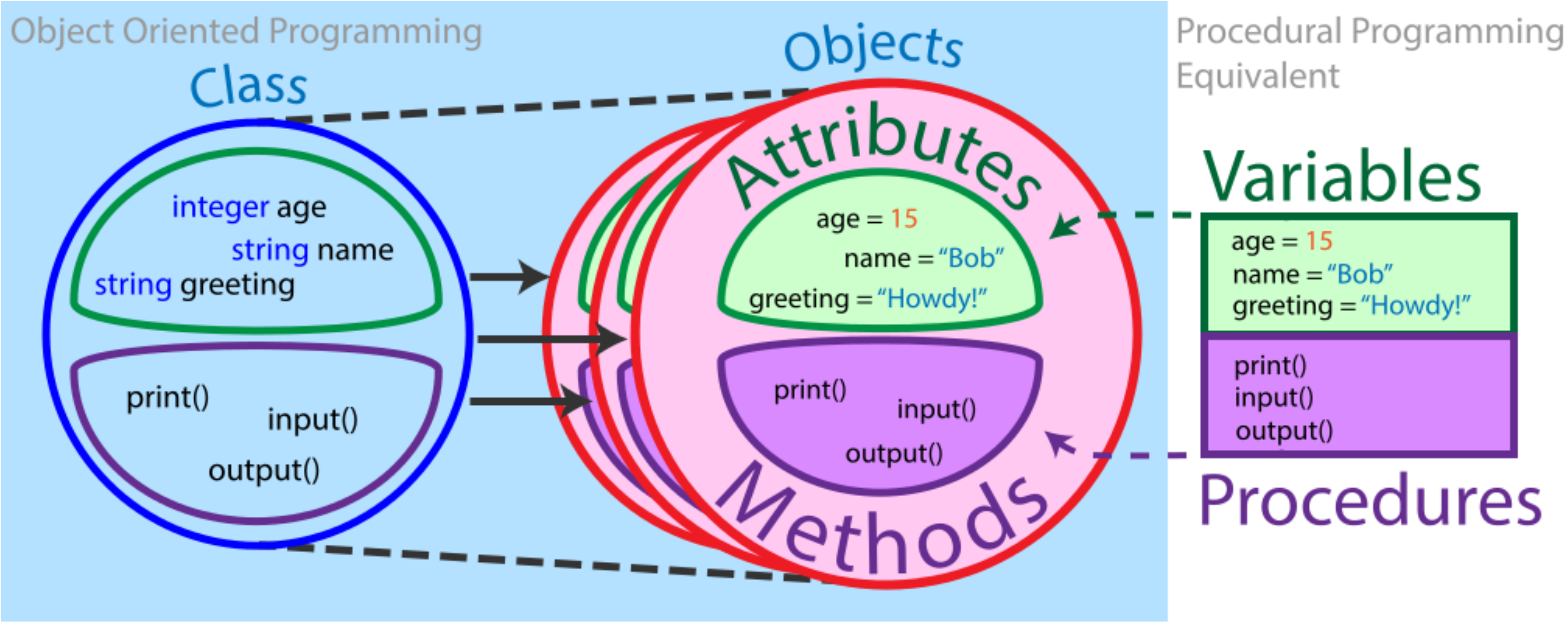
Coding Example: Hangman

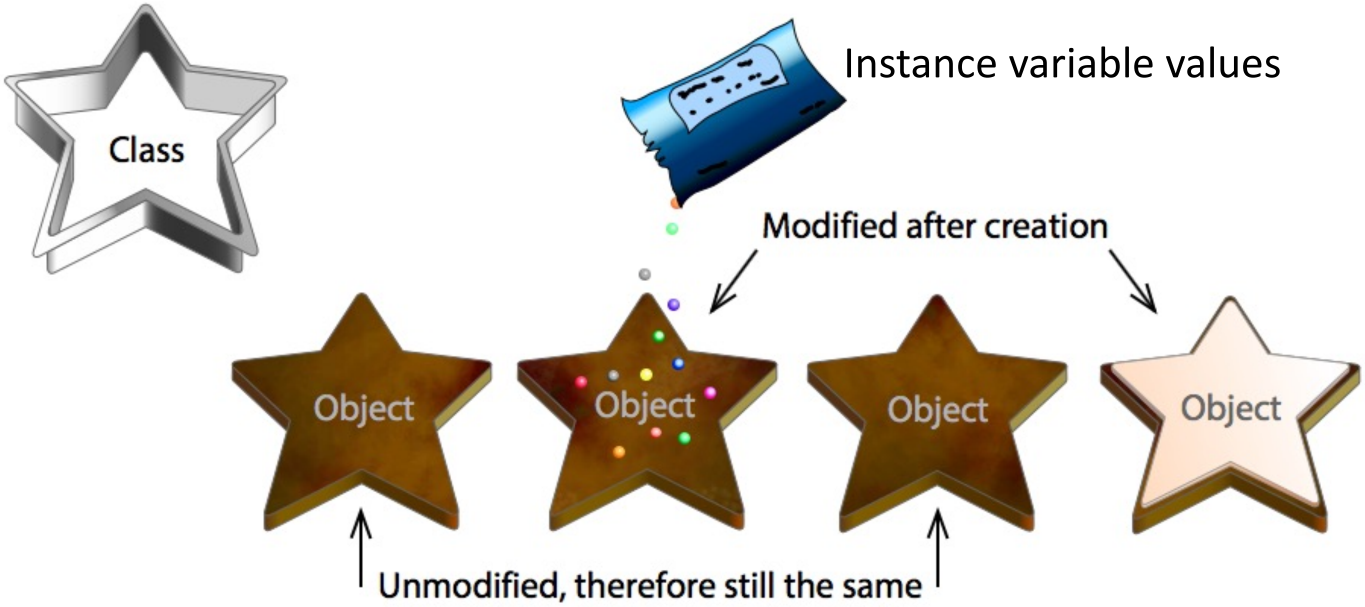
Create a text-based game that will allow the user to guess the letters in an unknown word. The word is picked randomly from a pre-defined list of available words. The word should be displayed before each guess, with blank entries for letters that have not yet been guessed. If the player guesses all the letters in the word before 6 incorrect guesses, they win, otherwise they lose.

Lab 3 Assignment: Mastermind

- Topics covered:
  - Loops and break statements
  - User input
  - f-strings
  - Basic list operations
  - List comprehension: embedded loops and conditionals
  - Code development process

Object Oriented vs. Procedural Code





Classes & Objects

- A class describes a set of *attributes* (class variables) and *methods* (class functions) available to every object instantiated from the class
  - By convention use PascalCase for class names, e.g.: `MyClass`

- An object is an *instance* of a class
  - Instance variable values can differ between objects of the same class
  - Objects have access to the variables and methods of their class
  - By convention use camelCase for object names, e.g.: `myObject`

Why OOP?

- Classes/objects abstract the design process
  - Bundle data (variables) and procedures (methods) together
  - provides a more logical structure to your code
  - makes large programs easier to plan and implement

- Classes can hide their internals from other programmers

- Extend classes through inheritance to create new classes with expanded functionality as needs grow

- Easy to re-use classes across multiple codes

- Maintaining object-oriented code is easier than procedural code since changes are fully modular

Almost Everything is an Object in Python

- `[0,1,2,3,2,2].count(2)`
- `{'a':1, 'b':2, 'c':3}.keys()`
- `"abcd".capitalize()`
- `(1.2).__add__(3.45)`
- `print.__doc__`

Class Example

```
class Student:

  'student class definition'   # optional class documentation string

  count = 0   # class variable (shared by all instances)

  # Magic methods:
  # constructor (optional, called when instantiating a new object):
  def __init__(self, first, last, grade=100):
    self.first = first    # instance variables
    self.last = last
    self.grade = grade
    Student.count += 1    # increment for each new instance
  # destructor (optional, called on deletion via 'del object'):
  def __del__(self):
    Student.count -= 1    # increment for each new instance

  # Class methods:
  def displayStudent(self):
    print("Name:%s %s, Grade:%s" % (self.first, self.last, self.grade))
```

```python
# Create objects with defined instance variables:
s1 = Student("Abbie", "Normal", 0)
s2 = Student("Sam", "Spade", 100)

# Access class methods and variables:
s1.displayStudent()
s2.displayStudent()
print("# of students = %d" % Student.count)

# Change instance variables:
s1.first = "Abby"
s1.grade = 100
s1.displayStudent()

# Delete an instance:
del s2
print("# of students after deletion = %d" % Student.count)
```

Built-In Class Attributes

- Access as: `ClassName.__attribute__`

`__dict__`	Dictionary with class namespace
`__doc__`	Class documentation string
`__name__`	Class name
`__module__`	Module name in which the class is defined (e.g. "__main__" or name of imported module)
`__bases__`	Tuple of base classes, in the order of their occurrence in the base class list

Class Inheritance

```
class ParentClass:
  def __init__():
    # instantiation code

class ChildClass(ParentClass)   # class name as argument
  def __init__():
    super().__init__()   # call parent init method, alternately:
                         # ParentClass.__init__()
    # instantiation code...
```

- The child class inherits all class & instance variables & methods
- New variables & methods specific to the child class can be added

Class Inheritance Example

```
class Employee:
  def __init__(self, id, name):
    self.id = id
    self.name = name

class SalaryEmployee(Employee): # Inherit from Employee class
  def __init__(self, id, name, weekly_salary):
    super().__init__(id, name)
    self.weekly_salary = weekly_salary
  def calculate_payroll(self):
    return self.weekly_salary

class HourlyEmployee(Employee): # Inherit from Employee class
  def __init__(self, id, name, hours_worked, hour_rate):
    super().__init__(id, name)
    self.hours_worked = hours_worked
    self.hour_rate = hour_rate
  def calculate_payroll(self):
    return self.hours_worked * self.hour_rate

class CommissionEmployee(SalaryEmployee): # Inherit from SalaryEmployee
class
  def __init__(self, id, name, weekly_salary, commission):
    super().__init__(id, name, weekly_salary)
    self.commission = commission
  def calculate_payroll(self):
    fixed = super().calculate_payroll()
    return fixed + self.commission
```

Class Composition

- Inheritance: “is a”
  - inherit from a base class
- Composition: “has a”
  - establish relationships between classes through instance variables referencing other objects

```
class Vehicle:
  def __init__(self, num_wheels, fuel_type):
    self.num_wheels = num_wheels
    self.fuel_type = fuel_type
    self.mileage = 0
  def report_mileage(self):
    return f"The vehicle has traveled {self.mileage} miles"

class MarsRover:
  def __init__(self, name, landing_date, num_wheels, fuel_type):
    self.vehicle = Vehicle(num_wheels, fuel_type) # extend via composition
    self.landing_date = landing_date

theRover = MarsRover("Perserverence", "02/18/2021", 6, "plutonium-238")
print(theRover.vehicle.report_mileage())
```

Public vs. Private Class Variables & Methods

- Private class variables and methods are accessible only within the class.
- Defined by prepending “__” to the name*:

```
class MyClass:
  __x = -1
  def myPublicMethod(self):
    print('public method called')
    self.__myPrivateMethod()
  def __myPrivateMethod(self):
    print('private method called’)
    print(__x)

c = MyClass()         # instantiate the class
c.myPublicMethod()    # prints all lines
c.myPrivateMethod()   # AttributeError (method is private)
print(c.__x)          # AttributeError (__x is private)
```

* The Python interpreter takes attributes (variables/methods) starting with “__” and renames them as “_classname__attribute”

Magic (Dunder) Methods

`__init__()`	Called when instantiating the class
`__iter__()`	Defines iteration behavior
`__str__()`	Dictates output of `print(ClassName)`
`__setitem__()`	Invoked when assigning to a dictionary
`__getitem__()`	Invoked when accessing value with a dict key
`__delitem__()`	Invoked when deleting item from a dict
`__len__()`	Defines output of `len(ClassName)`
`__contains__()`	Defines behavior of `in` operator
`__add__()`	Defines result of + operator
`__iadd__()`	Defines result of += operator

and many more…