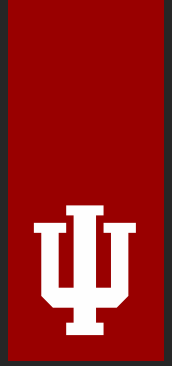


E210 Engineering Cyber-Physical Systems (Spring 2021)

Python Classes

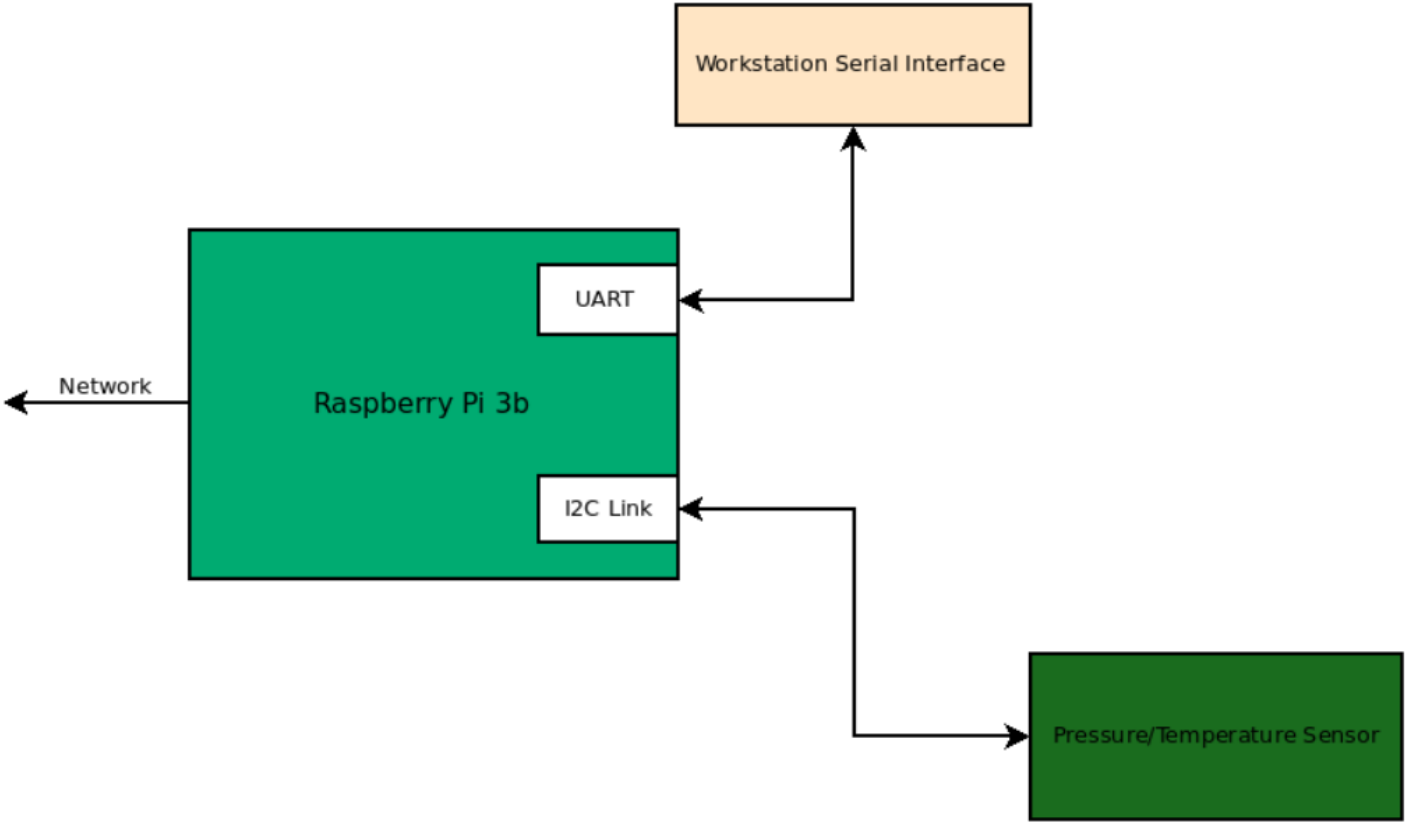
Bryce Himebaugh

Weekly Focus	Reading	Monday	Wed	Lab
Exam/CPS Introduction	Ref 1 Chapter 1	**3/8:** Exam 1	**3/10:** CPS Introduction	**Project 5 Raspberry PI Setup**
Raspberry Pi	Ref 2 Chapter 1-3	**3/15:** Pi Intro/UART Bus	**3/17:** Git/Github	
I2C Bus	Ref 3	**3/22:** I2C Bus	**3/24:** Wellness Day	**Project 6 I2C Pressure Sensor**
Python/Sensor	Ref 4, Ref 5	**3/29:** Classes/Modules	**3/31:** Pressure Sensor	
SPI	Ref 6	**4/5:** SPI Bus Overview	**4/7:** SPI HDL Design	**Project 7 SPI Connected I/O**
SPI	Ref 7 Chapter 1	**4/12:** SPI HDL Design	**4/14:** Sensor Memory	
Network Interface	Ref 7 Chapter 2	**4/19:** Ethernet Interface	**4/21:** MQTT	**Project 8 Network Interface**
MQTT/Flask	Ref 7 Chapter 14	**4/26:** Flask	**4/29:** Open Topic	

Final Exam Tues 5/4 10:10-12:10

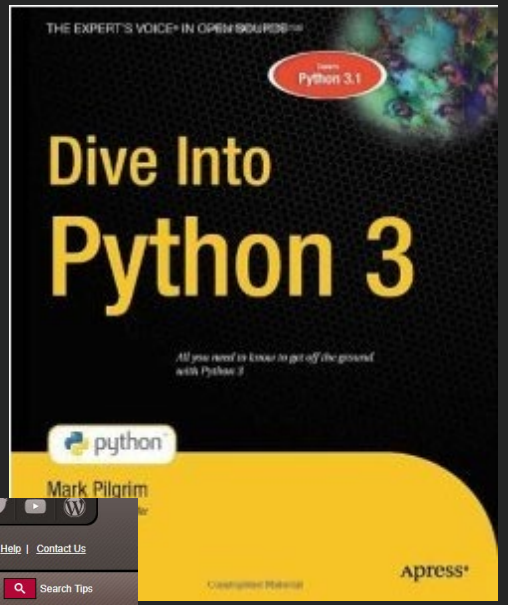
https://engr210.github.io/

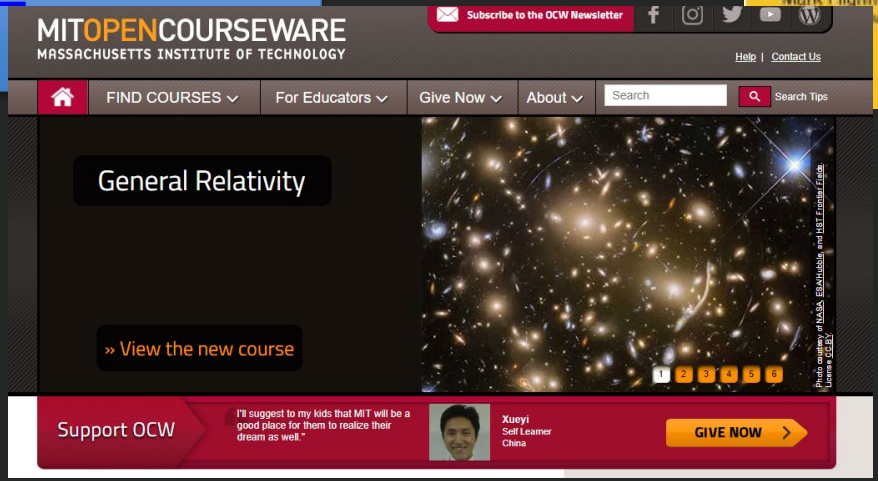
Raspberry I2C Link



Python Reference

- https://diveintopython3.net/
- MIT Open Courseware





MIT Open Courseware
Object Oriented Programming

OBJECT ORIENTED PROGRAMMING

OBJECTS

- Python supports many different kinds of data

```
1234      3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a string

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just "forget" about them
  - python system will reclaim destroyed or inaccessible objects – called "garbage collection"

WHAT ARE OBJECTS?

- objects are **a data abstraction** that captures…

(1) an **internal representation**

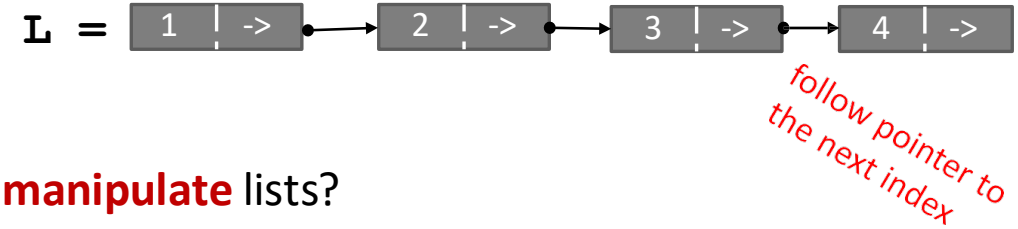
- through data attributes

(2) an **interface** for interacting with object

- through methods (aka procedures/functions)
- defines behaviors but hides implementation

EXAMPLE: [1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

ADVANTAGES OF OOP

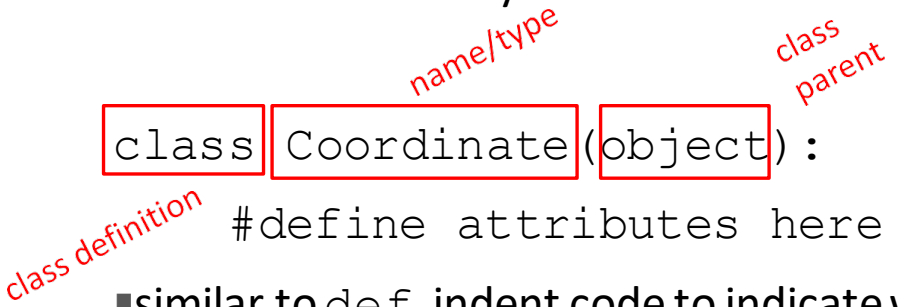
- **bundle data into packages** together with procedures that work on them through well-defined interfaces
- **divide-and-conquer** development
  - implement and test behavior of each class separately
  - increased modularity reduces complexity
- classes make it easy to **reuse** code
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- make a distinction between **creating a class** and **using an instance** of the class
- **creating** the class involves
  - defining the class name
  - defining class attributes
  - *for example, someone wrote code to implement a list class*
- **using** the class involves
  - creating new **instances** of objects
  - doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type



- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance next lecture)
  - `Coordinate` is a subclass of `object`
  - `object` is a superclass of `Coordinate`

WHAT ARE ATTRIBUTES?

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called** `__init__` to initialize some data attributes

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

special method to create an instance — is double underscore

what data initializes a `Coordinate` object

parameter to refer to an instance of the class

two data attributes for every `Coordinate` object

ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use `self` as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class  Coordinate(object):
    def__init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

THE POWER OF OOP

- **bundle together objects** that share
  - common attributes and
  - procedures that operate on those attributes
- use **abstraction** to make a distinction between how to implement an object vs how to use the object
- build **layers** of object abstractions that inherit behaviors from other classes of objects
- create our **own classes of objects** on top of Python's basic classes

Ips331 Class

lps331ap.py module

```
#!/usr/bin/env python3

import smbus
import sys
import time
import numpy as np

class lps331:
    ''' allows connection from Raspberry pi to I2C connected lps331 '''
```

lps331ap.py module

```
if __name__ == "__main__":
    sensor = lps331(1)
    print("Temperature = %0.2f Deg C "%(sensor.read_temperature()))
    print("Pressure = %0.2f inHg"%(sensor.read_pressure()))
    sensor.close()
```

lps331 class

```
def __init__(self,raspberry_pi_i2c_port=1):
        self.i2c_port_number = raspberry_pi_i2c_port
        self.bus = smbus.SMBus(self.i2c_port_number)
        self.address = self.find_sensor()
        if (self.address == -1):
            print("Error: could not read from sensor at i2c address 0x5d")
            sys.exit()
        self.enable_sensor()
```

Methods to Create …

```
    def find_sensor(self):
        ''' read the whoami byte from i2c address 0x5d and confirm to be 0xbb '''
        # Return the address if found (0x5d) and 0 if not found

        # @@@@ Your Code Here @@@@

        return(0);   # if the sensor was not located on either bus, return -1

    def i2c_address(self):
        return(self.address)

    def sample_once(self):
        ''' Cause the sensor to sample one time '''

        # @@@@ Your Code Here @@@@

        pass

    def read_temperature(self):
        ''' Sample, read temperature registers, and convert to inhg '''
        tempC = 0

        # @@@@ Your Code Here @@@@

        return(tempC)
```

Methods to Create …

```
def read_pressure(self):
    ''' Sample, read pressure registers, and convert to inhg '''
    press_inhg = 0

    # @@@@ Your Code Here @@@@

    return(press_inhg)

def enable_sensor(self):
    ''' Turn on sensor in control register 1'''

    # @@@@ Your Code Here @@@@

    pass

def disable_sensor(self):
  ''' Turn off sensor in control register 1 '''

    # @@@@ Your Code Here @@@@

    pass
```

Using the lps331ap.py Module

Importing the lps331ap Module

```
#!/usr/bin/env python3

import lps331ap

pt_sensor = lps331ap.lps331(1)
print("Temperature = %0.2f Deg C "%(pt_sensor.read_temperature()))
print("Pressure = %0.2f inHg"%(pt_sensor.read_pressure()))
pt_sensor.close()
```