CCPS109 Computer Science I L8-9

Lecturer: Nhan Tran

nhantran@Ryerson.ca

Agenda

Announcement

Lecture:

Debugging,

Exception,

Assertions

Object

CLASSES OF TESTS

- **Unit testing**
  - validate each piece of program
  - **testing each function** separately
- **Regression testing**
  - add test for bugs as you find them
  - **catch reintroduced** errors that were previously fixed
- **Integration testing**
  - does **overall program** work?
  - tend to rush to do this

DEBUGGING

- steep learning curve
- goal is to have a bug-free program
- tools
  - **built in** to IDLE and Anaconda
  - **Python Tutor**
  - `print` statement
  - use your brain, be **systematic** in your hunt

PRINT STATEMENT

- good way to **test hypothesis**
- when to print
  - enter function
  - parameters
  - function results
- use **bisection method**
  - put print halfway in code
  - decide where bug may be depending on values

DEBUGGING STEPS

- **study** program code
  - don’t ask what is wrong
  - ask how did I get the unexpected result
  - is it part of a family?

- **scientific method**
  - study available data
  - form hypothesis
  - repeatable experiments
  - pick simplest input to test with

ERROR MESSAGES - EASY

- trying to access beyond the limits of a list

  `test = [1,2,3]` then `test[4]` → `IndexError`

- trying to convert an inappropriate type

  `int(test)` → `TypeError`

- referencing a non-existent variable

  `a` → `NameError`

- mixing data types without appropriate coercion

  `'3'/4` → `TypeError`

- forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)
```

→ `SyntaxError`

LOGIC ERRORS - HARD

- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
  - someone else
  - a rubber ducky

DON'T

- Write entire program
- Test entire program
- Debug entire program

DO

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration testing ***

DON'T

- Write entire program
- Test entire program
- Debug entire program

DO

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration testing ***

DON'T

- Change code
- Remember where bug was
- Test code
- Forget where bug was or what change you made
- Panic

DO

- Backup code
- Change code
- Write down potential bug in a comment
- Test code
- Compare new version with old version

EXCEPTIONS AND ASSERTIONS

- what happens when procedure execution hits an **unexpected condition**?
- get an **exception**… to what was expected
  - trying to access beyond list limits

```
test = [1,7,4]

test[4]
```
→ **IndexError**

  - trying to convert an inappropriate type

```
int(test)
```
→ **TypeError**

  - referencing a non-existing variable

```
a
```
→ **NameError**

  - mixing data types without coercion

```
'a'/4
```
→ **TypeError**

OTHER TYPES OF EXCEPTIONS

- already seen common error types:
  - `SyntaxError`: Python can't parse program
  - `NameError`: local or global name not found
  - `AttributeError`: attribute reference fails
  - `TypeError`: operand doesn't have correct type
  - `ValueError`: operand type okay, but value is illegal
  - `IOError`: IO system reports malfunction (e.g. file not found)

DEALING WITH EXCEPTIONS

- Python code can provide **handlers** for exceptions

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

- exceptions **raised** by any statement in body of **`try`** are **handled** by the **`except`** statement and execution continues with the body of the `except` statement

HANDING SPECIFIC EXCEPTIONS

- have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))  print("a/b =
    ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:  print("Can't divide
    by zero")
except:
    print("Something went very wrong.")
```

HANDING SPECIFIC EXCEPTIONS

- have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute if these errors come up

HANDING SPECIFIC EXCEPTIONS

- have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

for all other errors

HANDING SPECIFIC EXCEPTIONS

- have **separate `except` clauses** to deal with a particular type of exception

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors

OTHER EXCEPTIONS

- `else:`
  - body of this is executed when execution of associated `try` body **completes with no exceptions**
- `finally`:
  - body of this is **always executed** after `try`, `else` and `except` clauses,
    - even if they raised another error or executed a `break`, `continue` or `return`
  - useful for clean-up code that should be run no matter what else happened (e.g. close a file)

WHAT TO DO WITH EXCEPTIONS?

- what to do when encounter an error?
- **fail silently**:
  - substitute default values or just continue
  - bad idea! user gets no warning
- return an **"error" value**
  - what value to choose?
  - complicates code having to check for a special value
- stop execution, **signal error** condition
  - in Python: **raise an exception**

```
raise Exception("descriptive string")
```

EXCEPTIONS AS CONTROL FLOW

- don't return special values when an error occurred and then check whether 'error value' was returned
- instead, **raise an exception** when unable to produce a result consistent with function's specification

```
raise <exceptionName>(<arguments>)
```

EXCEPTIONS AS CONTROL FLOW

- don’t return special values when an error occurred and then check whether ‘error value’ was returned
- instead, **raise an exception** when unable to produce a result consistent with function’s specification

```
raise <exceptionName>(<arguments>)
```

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

EXAMPLE: RAISING AN EXCEPTION

```
def get_ratios(L1, L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i] """
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/L2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = not a number
        except:
            raise ValueError('get_ratios called with bad arg')
    return ratios
```

EXAMPLE: RAISING AN EXCEPTION

```
def get_ratios(L1, L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i] """
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/L2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = not a number
        except:
            raise ValueError('get_ratios called with bad arg')
    return ratios
```

manage flow of program by raising own error

EXAMPLE OF EXCEPTIONS

- assume we are **given a class list** for a subject: each entry is a list of two parts
  - a list of first and last name for a student
  - a list of grades on assignments

```
test_grades = [[['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
[['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

EXAMPLE CODE

```
[[['peter', 'parker'], [80.0, 70.0, 85.0]],
[['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

```
def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    return new_stats

def avg(grades):
    return sum(grades)/len(grades)
```

ERROR IF NO GRADE FOR A STUDENT

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
               [['captain', 'america'], [8.0,10.0,96.0]],  [['deadpool'],
               []]]
```

- get `ZeroDivisionError: float division by zero` because try to `return sum(grades)/len(grades)`

ERROR IF NO GRADE FOR A STUDENT

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
               [['captain', 'america'], [8.0,10.0,96.0]],
               [['deadpool'], []]]
```

- get `ZeroDivisionError: float division by zero` because try to

```
return sum(grades)/len(grades)
```

length is 0

OPTION 1: FLG AN ERROR BY PRINTING A MESSAGE

- decide to **notify** that something went wrong with a msg

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```

- running on some test data gives

```
warning: no grades data
```

flagged the error

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], None]]
```

because `avg` did not return anything in the except

OPTION 2: CHANGE THE POLICY

- decide that a student with no grades gets a **zero**

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return 0.0
```

- running on some test data gives

```
warning: no grades data
```

still flag the error

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

now avg returns 0

ASSERTIONS

- want to be sure that **assumptions** on state of computation are as expected
- use an **`assert` statement** to raise an `AssertionError` exception if assumptions not met
- an example of good **defensive programming**

EXAMPLE

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

function ends immediately if assertion not met

- raises an `AssertionError` if it is given an empty list for grades
- otherwise runs ok

ASSERTION AS DEFENSIVE PROGRAMMING

- assertions don't allow a programmer to control response to unexpected conditions
- ensure that **execution halts** whenever an expected condition is not met
- typically used to **check inputs** to functions, but can be used anywhere
- can be used to **check outputs** of a function to avoid propagating bad values
- can make it easier to locate a source of a bug

WHERE TO USE ASSERTIONS?

- goal is to spot bugs as soon as introduced and make clear where they happened
- use as a **supplement** to testing
- raise **exceptions** if users supplies **bad data input**
- use **assertions** to
  - check **types** of arguments or values
  - check that **invariants** on data structures are met
  - check **constraints** on return values
  - check for **violations** of constraints on procedure (e.g. no duplicates in a list)

Object Oriented Programming

OBJECTS

- Python supports many different kinds of data

`1234`   `3.14159`   `"Hello"`   `[1, 5, 7, 11, 13]`

`{"CA": "Canada", "JP": "Japan"}`

- 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**

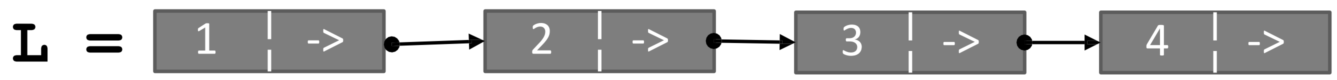
- through data attributes

(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

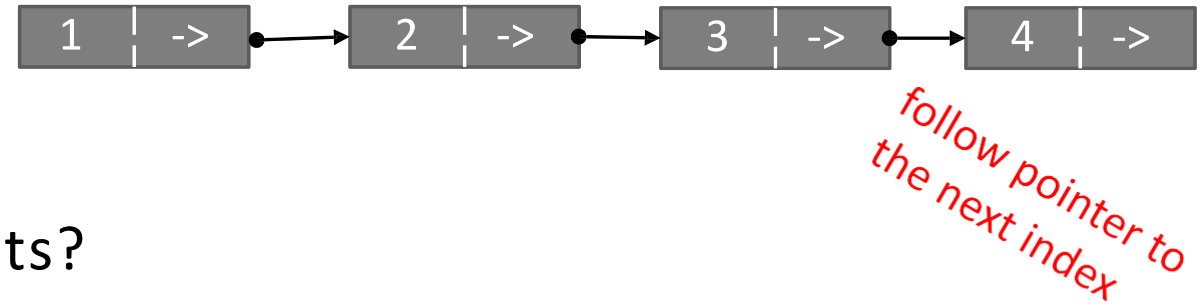
- how are lists **represented internally**? linked list of cells



follow pointer to the next index

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

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

L = 

- 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)`

Baking Cookies

Recipe

Ingredients

- 1 cup butter, softened
- 1 cup white sugar
- 1 cup packed brown sugar
- 2 eggs
- 2 teaspoons vanilla extract
- 1 teaspoon baking soda
- 2 teaspoons hot water
- ½ teaspoon salt
- 3 cups all-purpose flour
- 2 cups semisweet chocolate chips
- 1 cup chopped walnuts

Method:

Step 1

Preheat oven to 350 degrees F (175 degrees C).

Step 2

Cream together the butter, white sugar, and brown sugar until smooth. Beat in the eggs one at a time, then stir in the vanilla. Dissolve baking soda in hot water. Add to batter along with salt. Stir in flour, chocolate chips, and nuts. Drop by large spoonfuls onto ungreased pans.

Step 3

Bake for about 10 minutes in the preheated oven, or until edges are nicely browned.

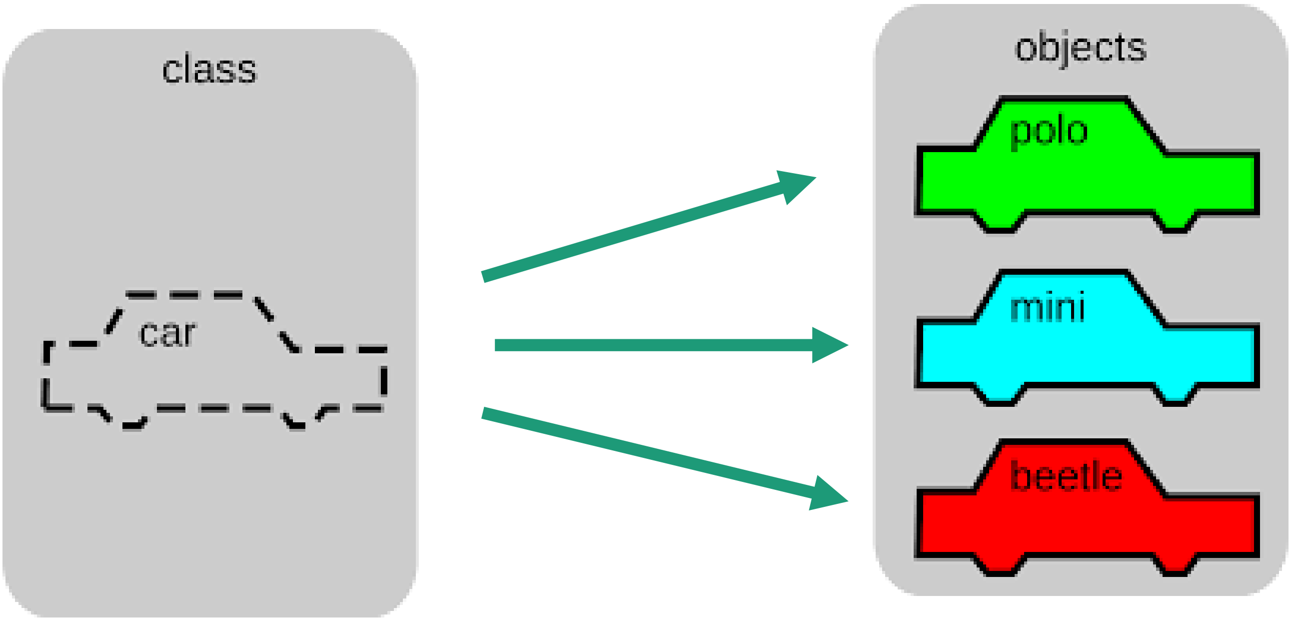
Recipe

Recipes are classes

And things (cookies) they make are

Object

Or



DEFINE YOUR OWN TYPES

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

```
class Coordinate(object):
    #define attributes here
```

- 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`

DEFINE YOUR OWN TYPES

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

name/type

class parent

```
class Coordinate(object):
    #define attributes here
```

class definition

- 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
```

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
```

what data initializes a Coordinate object

special method to create an instance is double __ underscore

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
```

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`)

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

WRAPPING YOUR HEAD AROUND TYPES AN CLASSES

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class

a Coordinate class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

SPECIAL OPERATORS

- ■ +, -, ==, <, >, len(), print, and many others
- https://docs.python.org/3/reference/datamodel.html#basic-customization

■ like `print`, can override these to work with your class

■ define them with double underscores before/after

```
__add_(self, other)    →    self + other
__sub_(self, other)    →    self - other
__eq_(self, other)     →    self == other
__lt_(self, other)     →    self < other
__len__(self)          →    len(self)
__str__(self)          →    print self
```

... and others

EXAMPLE: FRACTIONS

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers
  - numerator
  - denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - add, subtract
  - print representation, convert to a float
  - invert the fraction
- the code for this is in the handout, check it out!

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

End….