Part II Self-defined Data Types -- Classes

- Python is an object-oriented programming language
- Key mechanism: classes
- Each data class is a set; each object is an element of a data type

  -- **int** is a type, 1, 2, 3,… are its objects.
  -- **str** is a type, 'hello' is an string instance

- A class include:

  -- attributes of the objects specified by its type
  -- methods (which are actually functions) for processing objects

```
import pandas as pd
import …
Import …

class Employee:
        …
…
class Salary:
        …

def  function1(…):
        …
…
def  function5(…):
        …

if __name__ == '__main__’:
      function 5(…)
```

```
import pandas as pd
import …
Import …

class Employee:
        …
…
class Salary:
        …

if __name__ == '__main__’:
      …
```

- Defining a class involves:

    -- naming the class

    -- specifying the class attributes

    -- defining methods

- For example, define a class rational_num
- How to use a class?

    -- declaring an object of the class;

    -- call a function on objects

```
#  defining a class
class  rational_num:
   “”” define attributes here”””
      def __init__(self, a, b):
         #create an instance
         self.numerator=a
         self.denum=b
      def __add__(self, another):
         # addition operator
```

name

argument

keyword

method name

```
r = Rational_num(2, 0.5)
```

- A=32+45
- Str="my"+" car "
- We can redefine __add__ to make addition operation work for rational numbers defined by us, This feature is called Operator Overloading

Special methods available for int

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other)
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

```
#  defining a class
class  Rational_num:
   # define attributes here


        def __init__(self, a, b):
                #create an instance
                self.numerator=a
                self.denum=b

        def __add__(self, another):
                # addition operator
                x= self.numerator + another.numerator
                y= self.denum + another.denum
                return Rational_num(x, y)

r1=Rational_num(1,1)
r2=Rational_num(3,1)
r=r1+r2;
print(r.numberator)
print(r.denum)
```

Special methods: __str__,

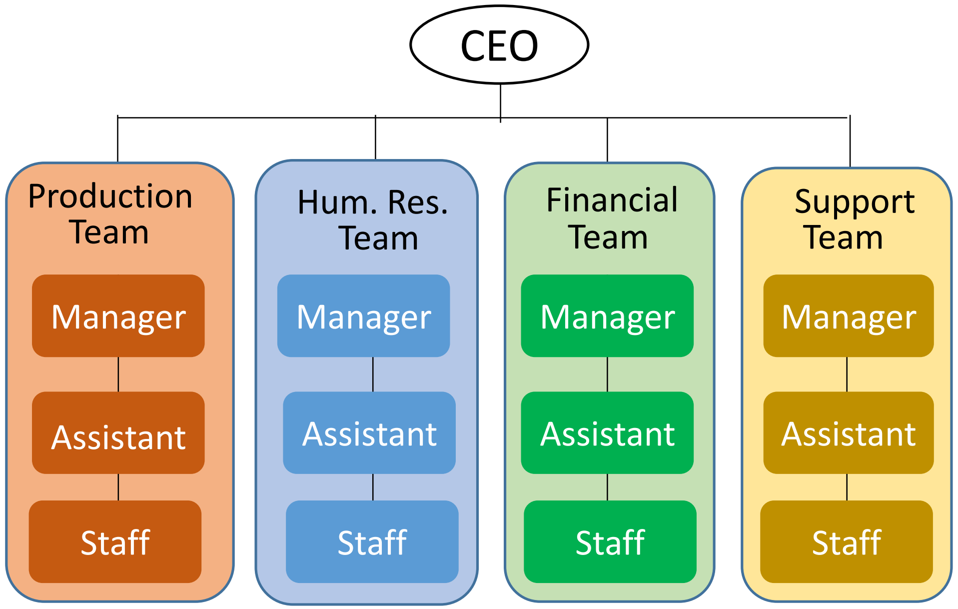
- We mentioned that __init__ is a special method for creating an object instance, called constructor.
- __str__ and __repr__ are called when printing an object;
- __ is pronounced "d(ouple)under"

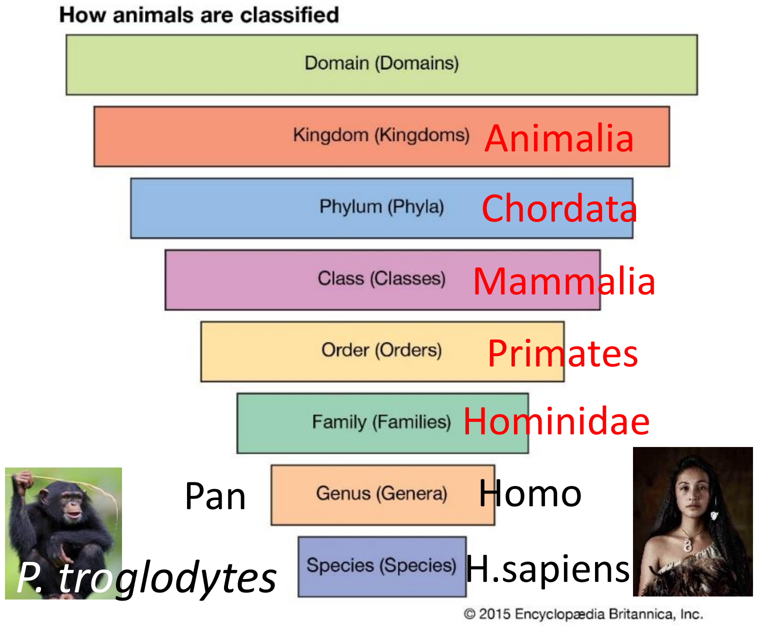
```
class  Employee:
    raise_percent= 1.04
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
    def apply_raise(self):
        self.pay=int (self.pay * self.raise_amt)
    def __str__(self):
        return f'{self.first} {self.last}-{self.pay}'
```

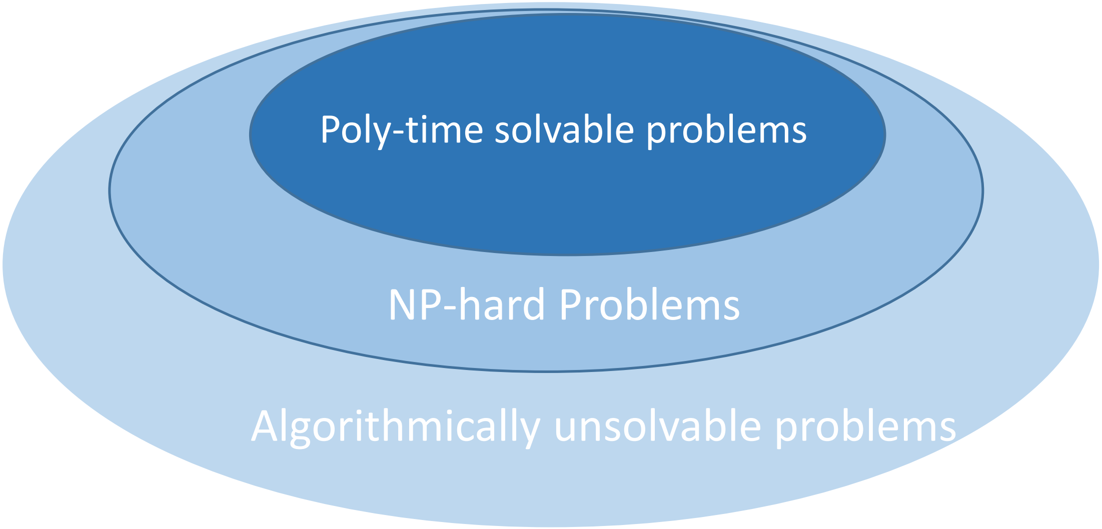
```
dep1=Employee('John', 'Doe')
print(dep1)
```

Hierarchical Structures

- Hierarchical classification thinking is a key approach in scientific study and human resource management







- Algorithmic unsolvable problems
  - -- Check whether or not a Python program exist without error
  - -- Check whether or not a multivariable polynomial equation has an integer solution
  - -- Given a set of 6 or more 3x3 matrices, check whether they can be multiplied in some order, possibly with repetition, to yield the zero matrix
- NP-hard problems
  - -- Given a set of integers, is it possible to divide it into two parts so that the sums of integers for the two parts are equal?
  - -- Hamiltonian cycle problem: given a graph, is there a cycle that passes each vertex exactly once?
- Liner-time solvable problem

  Big data problems

Class and Super-class

- The argument **rational_num** suggests

  -- rational_num is a superclass of even_num

  -- even_num is a subclass of retional_num

- Semantically, the objects of even_num inherits all its attributes defined in the rational_num class.
- This feature enhances the clarity, robust and re-usability.

```
# defining a class

class even_num (rational_num):
   #define attributes here
       def __init__(self, a):
              #create an instance
              self.numerator=a
              self.denum=1
```

keyword name parent class

```
a=even_num(2)
b=even_num(4)
c=a+b
print(a+b)
```

Inherited operator “+”

```
#  defining a class
class  Employee:
        raise_percent= 1.04
        def __init__(self, str1, str2, pay):
                self.firstName = str1
                self.lastName = str2
                self.pay = pay
        def fullname(self):
              pass
        def apply_raise(self):
              self.pay=int (self.pay * self.raise_percent)
class Data_Scientist(Employee):
          pass
```

class variable

```
assistant = Employee (“Joe”, “Doe”, 5000)
assistant.apply_raise()
print(assistant.pay)
Smith = Data_Scientist (“John”, “Smith”, 5000)
Smith.apply_raise()
print(Smith.pay)
```

```
#  defining a class
class  Employee:
         raise_percent= 1.04
        def __init__(self, str1, str2, pay):
                self.firstName = str1
                self.lastName = str2
                self.pay = pay
        def fullname(self):
              pass
        def apply_raise(self):
              self.pay=int (self.pay * self.raise_percent)

class Data_Scientist (Employee):
          raise_percent=1.1
```

class variable modified

```
Smith =Data_Scientist (‘Joe’, ‘Smith’, 5000)
Smith.apply_raise()
print(Smith.pay)
```

```
#  defining a class
class  Employee:
         raise_percent= 1.04

        def __init__(self, str1, str2, pay):
                self.firstName = str1
                self.lastName = str2
                self.pay = pay

        def fullname(self):
                pass
        def apply_raise(self):
                self.pay=int (self.pay * self.raise_percent)

class DataScientist(Employee):
        raise_percent=1.1
        def __init__(self, first, last, pay, prog_lang):
              super().__init__(first, last, pay)
              self.prog_lang = prog_lang
```

```
dep1 = DataScientist('Joe', 'Smith', 5000, 'Python')
dep1.apply_raise()
print(dep1.prog_lang)
```

- As a subclass of Employee, DataScientist inherits all features, variables and functions defined in Employee
- Over-writing inherited functions and variables are allowed

  for customization

- Important concepts

-- Regular, class and static methods

-- Property decorator: getter, setter and delete

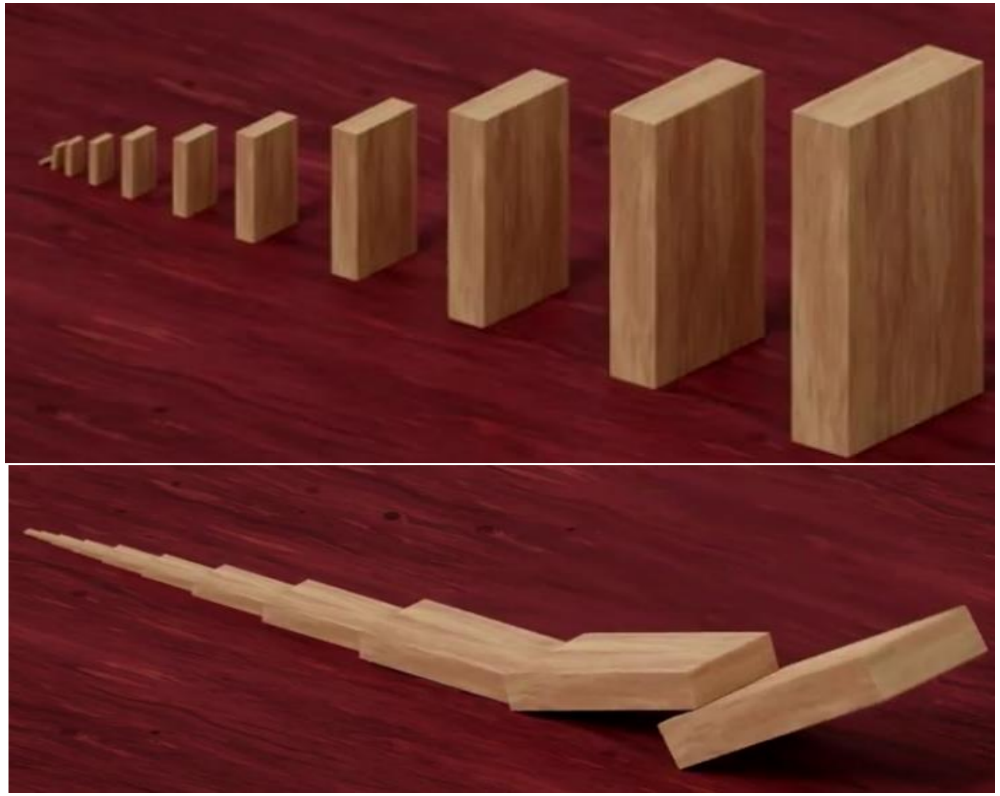
https://www.youtube.com/watch?v=ZDa-Z5JzLYM&t=6s

https://www.youtube.com/watch?v=BJ-VvGyQxho

https://www.youtube.com/watch?v=rq8cL2XMM5M

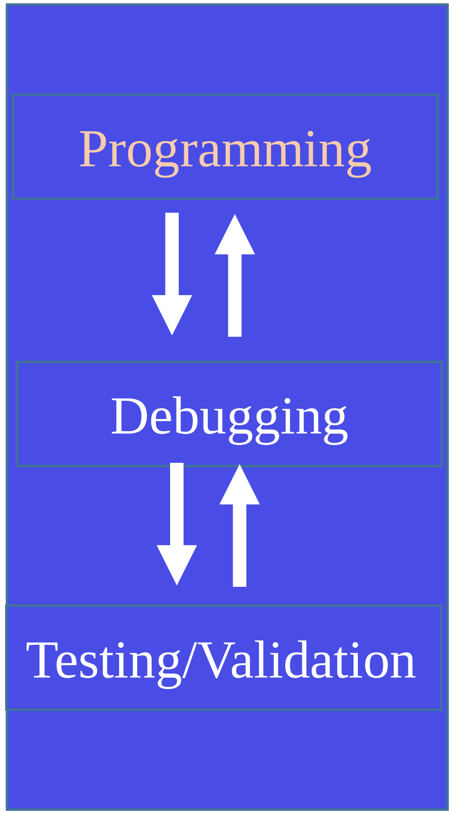
https://www.youtube.com/watch?v=RSl87lqOXDE

Part III Program testing and debugging



Expectation

Reality



- Use **abstraction and decomposition**
- Modularize programs
- Write **specifications** for functions
- Consider conditions on I/Os

Programming

Debugging

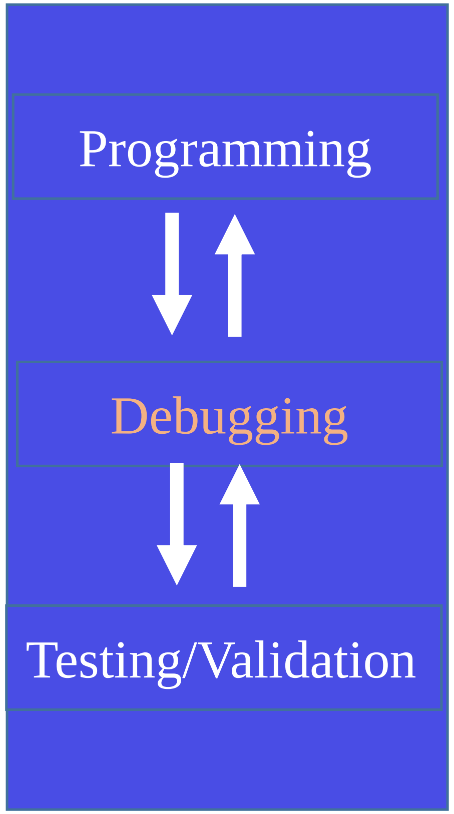
Testing/Validation

- Syntax errors
- Logical errors

  -- branches

  -- for loop

  -- while loop



- Grammar errors
- Logical errors
  - -- branches
  - -- for loop
  - -- while loop

Possible Errors

1. Branches

-- Conditions are complete or not

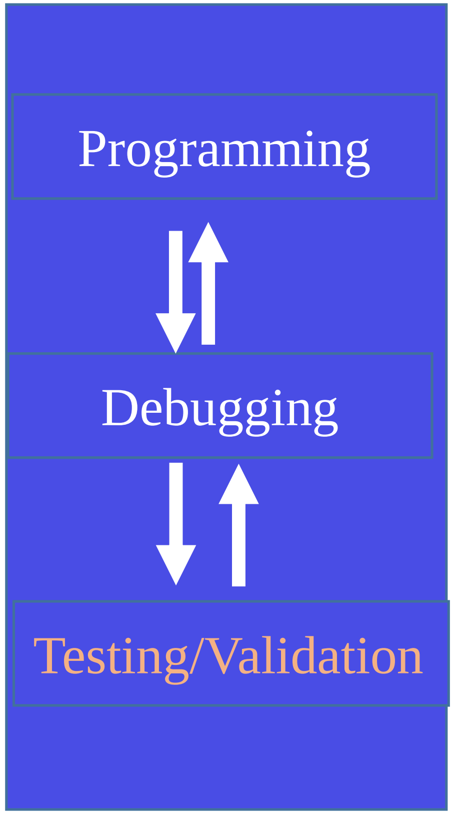
-- Conditions are correctly given

2. While/for loops

-- loop not entered

-- exit loop early

-- never exit loop



- Unit testing

  -- test each function/class separately

- Regression testing

  -- add test after each bug is fixed

  -- catch new errors arising from debugging

- Integration testing

  -- does the whole program work?

Testing Approaches

1. Black box testing

based on specification

2. Glass box testing

work through code

Black box testing

- Designed without looking at the code
- Better done by someone other than the programmer to avoid biases
- Consider boundary/trivial cases
- Test random cases

- For example, for functions defined on sets
They are likely bugs for empty sets.

- For integer functions, it should be tests on
odd, even, primer numbers, positive, negative numbers.

Glass box testing

- Use code directly to design test cases
- Consider boundary cases for branches, for/while loops

```
def abs(x):
 # compute the absolute value of x

    if  x<-1:
        return –x
    else:
       return x
```

Bad Habits

- Write entire program
- Debug/test entire program

Good Habits

- Write and test on function/call basis
- Then integration testing