Unit 1 – Python Basics

① 2 things every computer can do < Perform calculations / Remember results

② Declarative knowledge vs. Imperative knowledge
(statements of facts) (a recipe of "how-to")
Algorithm: a conceptual idea Program: a concrete instantiation of an algorithm

③ Turing 6 primitive operations: "move left; move right; scan; print; erase; do nothing"

④ Fixed program computers vs. Stored program computers (nowadays' computers)

⑤ A program counter: points the computer to the next instruction to execute in the program

⑥ The computer executes the instructions mostly in a linear sequence, except sometimes it jumps to a different place in the sequence.

⑦ Programming Language: provides a set of primitive operations
Expressions: complex but legal combinations of primitives in a programming language.

⑧ Errors < syntatic errors / Static semantic errors (Python check as the program runs) / logic errors (semantic errors)

Remark < static semantic – determine whether each line has at least 1 meaning / semantic – a specific meaning to each line.

⑨ Program < definitions evaluated / commands executed by Python intepreter in a shell
(commands (statements) instruct intepreter to do sth.)

⑩ Objects: programs manipulate data objects. They are either < Scalar: can NOT be subdivided / Non-scalar: have internal structure that can be accessed

Python < Scalar objects: int; float; bool (e.g. True); NoneType (None) / Non-scalar objects:

⑪ >>> 4>5 or 3<4 and 9>8 (True)

Operator Precedence

Operator	Description
:=	Assignment expression
lambda	Lambda expression
if – else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder [5]
+x, -x, ~x	Positive, negative, bitwise NOT
**	Exponentiation [6]
await x	Await expression
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...),	Binding or parenthesized expression, list display, dictionary display, set display
[expressions...], {key: value...}, {expressions...}	

⑫ >>> 5/2 == 5/2.0 (True) >>> 5*2 == 5.0*2.0 (True)

>>> 2**2 == 2.0**4 (False) Tricky Comparasions

⑬ String concatenation: Unlike Java, can NOT add str + int (illegal & traceback)

"len(str)" to check string length ; stringname[index] will show "index"th character

[-1] [a:b] [a:] [:a] [:] all acceptable

end indices NOT included

⑭ String multiplication:

e.g. >>> "bc" *3

"bcbcbc"

Unlike Java, we can compare contents of strings with "=="

⑮ Tricks of "print(XXX)"

Further, every print(XXX) creates a "\n" in the end

```
In [80]: print("my fav num is", x, ".", "x =", x)
my fav num is 1 . x = 1

In [81]: print("my fav num is" + x_str + "." + "x = " + x_str)
my fav num is1.x = 1

In [82]: print("my fav num s " + x_str + ". " + "x = " + x_str)
my fav num s 1. x = 1
```

comma in print() defaults a space

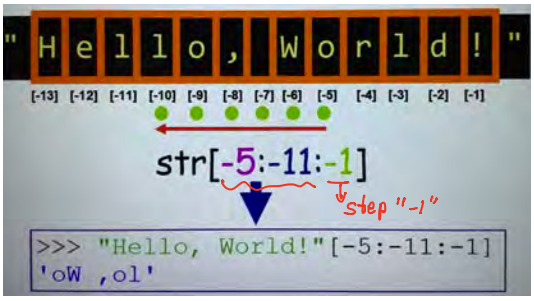
"+" no default

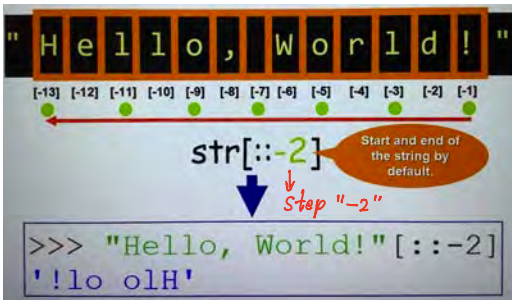
⑯ Tricks of range in [a:b] (advanced):

>>> str4 = "helloworld"

>>> str4[1:9:2] "elwr" range [1,9), every other char (step: +2) ∴ 1,3,5,7 indices

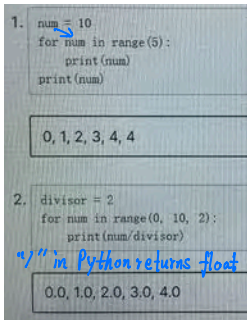
>>> str4[::-1] "dlrowolleh" range all, inverse order (note str4[::1] NOT work)





⑰ do NOT name a variable "sum"

⑱ Tricky Q



⑲ Casting: e.g. > a = int('4')

⑳ Binary expression: internally, computer represents numbers in binary.

in Python, the intepreter uses Floats to approximate real number

Express decimal number in binary way:

e.g. $19 = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 \rightarrow 10011$ Integer part

e.g. $3/8 = 0.375 = 0*2^{-1} + 1*2^{-2} + 1*2^{-3} \rightarrow 0.011$ Fractions part

(21) Convert Decimal to Binary in Python

CONVERTING DECIMAL INTEGER TO BINARY

- Consider example of
  - $x = 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 10011$
- If we take remainder relative to 2 (`x%2`) of this number, that gives us the last binary bit
- If we then divide x by 2 (`x//2`), all the bits get shifted right
  - $x//2 = 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 1001$
- Keep doing successive divisions; now remainder gets next bit, and so on
- Let's us convert to binary form

```
if num < 0:
    isNeg = True
    num = abs(num)
else:
    isNeg = False
result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
if isNeg:
    result = '-' + result
```

WHAT ABOUT FRACTIONS?

- $3/8 = 0.375 = 3*10^{-1} + 7*10^{-2} + 5*10^{-3}$
- So if we multiply by a power of 2 big enough to convert into a whole number, can then convert to binary, and then divide by the same power of 2
- 0.375 * (2**3) = 3 (decimal)
- Convert 3 to binary (now 11)
- Divide by 2**3 (shift right) to get 0.011 (binary)

```
x = float(input('Enter a decimal number between 0 and 1: '))

p = 0
while ((2**p)*x)%1 != 0:
    print('Remainder = ' + str((2**p)*x - int((2**p)*x)))
    p += 1

num = int(x*(2**p))

result = ''
if num == 0:
    result = '0'
while num > 0:
    result = str(num%2) + result
    num = num//2
for i in range(p - len(result)):
    result = '0' + result

result = result[0:-p] + '.' + result[-p:]
print('The binary representation of the decimal ' + str(x) + ' is
' + str(result))
```

※

remainder

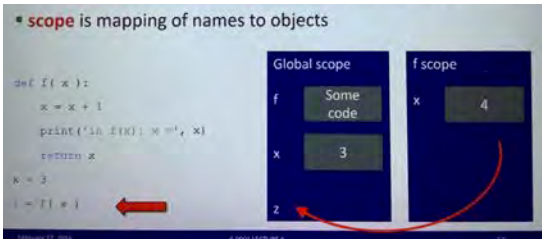
Total length

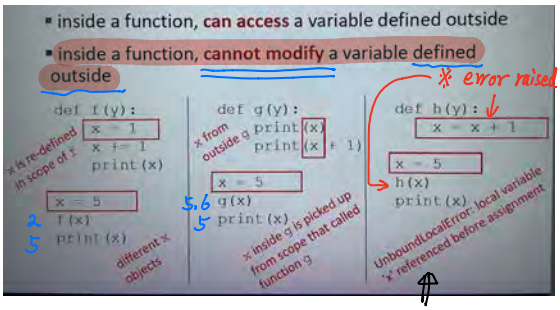
fill "0" in front fraction positions if necessary

Unit 2 – Simple Programs

Part A – Normal Functions

① Variable Scope



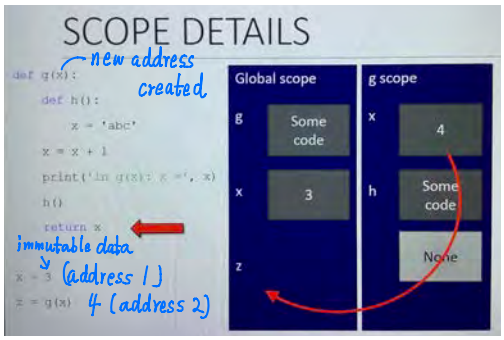


No error if pass by parameter!!

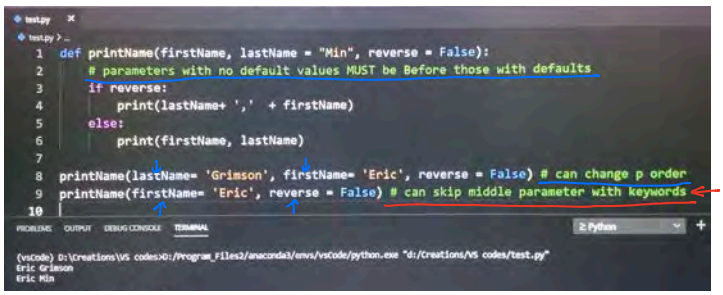
② Functions return "None" by default if no specific return statements

③ only ONE return executed inside a function

④ Parameters
- Immutable data → pass by value (make a separate copy)
- Mutable data → pass by reference (pointing to the original address)



⑤ Keyword argument And Default values



```
def printName(firstName, lastName = "Min", reverse = False):
    # parameters with no default values MUST be Before those with defaults
    if reverse:
        print(lastName+ ',' + firstName)
    else:
        print(firstName, lastName)

printName(lastName= 'Grimson', firstName= 'Eric', reverse = False) # can change p order
printName(firstName= 'Eric', reverse = False) # can skip middle parameter with keywords
```

```
(vsCode) D:\Creations\VS codes>D:/Program_Files2/anaconda3/envs/vsCode/python.exe "d:/Creations/VS codes/test.py"
Eric Grimson
Eric Min
```

※ this principle can be utilised to create objects missing some attributes

⑥ Doc String in each Function (not mandatory, but Recommended)
- Assumptions (inputs...)
- Guarantees (outputs...)

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print "hi"
    return i%2 == 0
```

Part B - Recursion & Recursive Functions

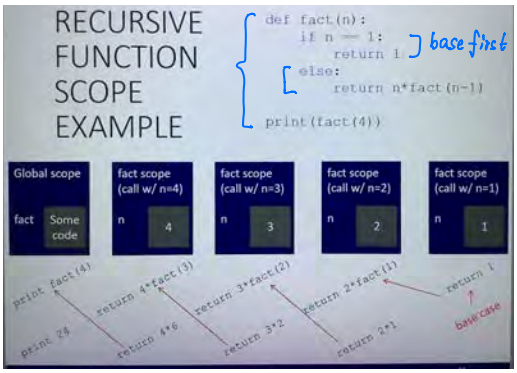
① idea: Divide-and-conquer / Decrease-and-conquer Go deepest and then come back

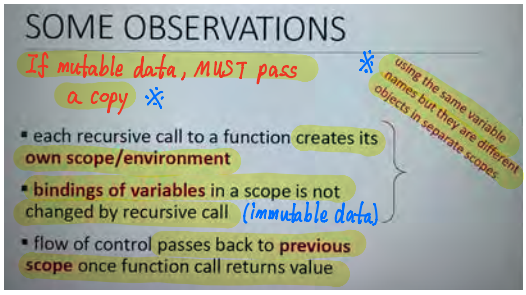
② Recursion: a function calls itself; In programming, goal is to NOT have indefinite recursion

- MUST have 1 or more base cases that are easy to solve
- MUST solve the same problem on some other input with the goal of simplifying the larger problem input

③ usually in recursive functions, we write lines for base cases first, recursive step(s) second.

④ Recursive Function Scope e.g.





⑤ Inductive Reasoning - ensure calling itself with a smaller parameter(s) scope
- base cases
- f(n-1) → f(n)

Think about Mathematical Induction

⑥ e.g. Towers of Hanoi

```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))

def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

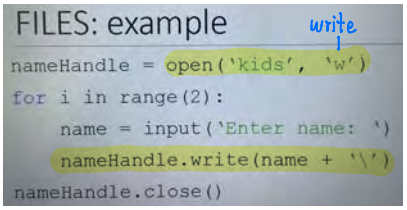
⑦ e.g. Palindrome (回文)

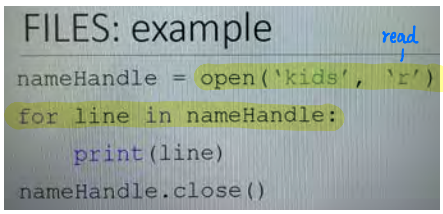
```
def isPalindrome(s):
    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans
    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])
    return isPal(toChars(s))
```

Part C - Modules and Files

① "from XX import *" may cause confusion. Python will always call current file's functions first.

② File handle Syntax: > <varName> = open(<fileName>, <charCommand>)





Unit 3 - Structured Types

Part A - Tuples

① Tuples can be added "+", and sliced "[a:b]", but can NOT be modified (NOT change address/value)

② Non-empty tuples MUST include at least ONE comma ","

③ Usage 1: Swap variable values e.g. > (x, y) = (y, x)

④ Usage 2: return more than 1 value at a time e.g. > return (x, y)

⑤ Usage 3: expand through concatenation "+" (make copies during the process)

⑥ Usage 4: Extract portion through [a:b] (make copies during the process)

⑦ range (a,b,s) is a special procedure that behaves like a tuple

Part B - Lists (Mutable)

① <list>.append(x) expand the original object (not new copy)

② concatenation "+" does NOT create new copies.

③ Delete element(s)
- specific index: del(L[index])
- at end of list: L.pop() (& return the last element)
- specific element: L.remove(element) (only the first found from left)
  - error if no such element

④ Convertion with Strings
- string to list with every character an element: > list(s) ※
- split a string to list with separater: > s.split(x)
  - if NO x, separate by spaces
- list of strings to a string: > 'x'.join(L)
  - (the list must NOT contain non-str element)
  - ↑ any symbol

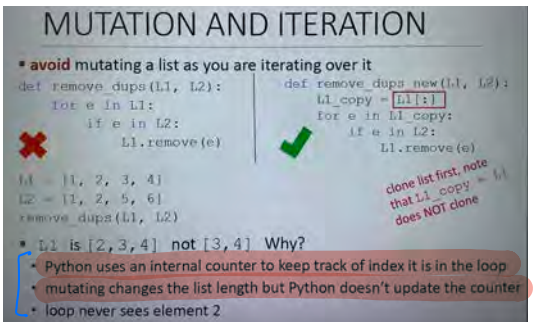
⑤ (exclude operators) Function calls vs. OO methods
- ↑ Create a new copy
- ↑ modify the original one

e.g.
- sorted(L) → returns a new sorted copy, NOT mutate L
- L.sort() → sort the L list directly

more in https://docs.python.org/2/tutorial/datastructures.html

⑥ Aliasing of mutable data → be careful about side effect
to avoid side effects, we usually clone the list to manipulate

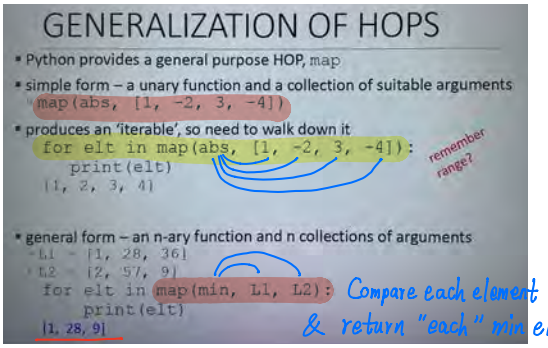
⑦ Avoid Looping with to be modified List(s)



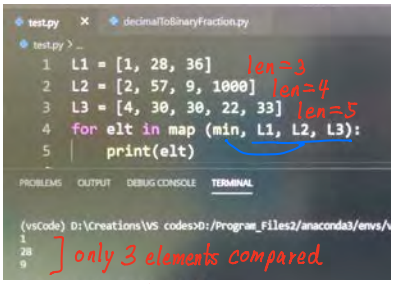
Part C – Functions as Objects; Dictionaries

① functions are first class objects < have types / can be elements of data structures like lists / can appear in expressions

② Generalization of Hops – map: n-ary function and n collections of arguments.



syntax: > map(<func>, *iterables)



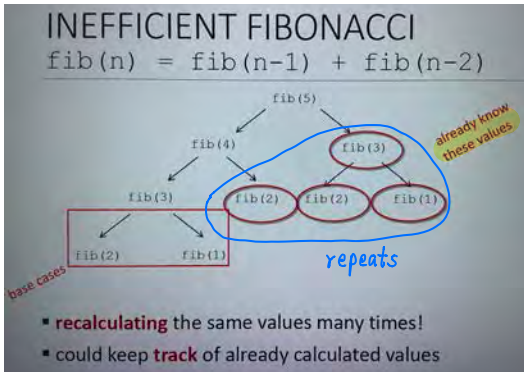
Remark: only compare the common indices' elements; the longer parts will not iterate at all.

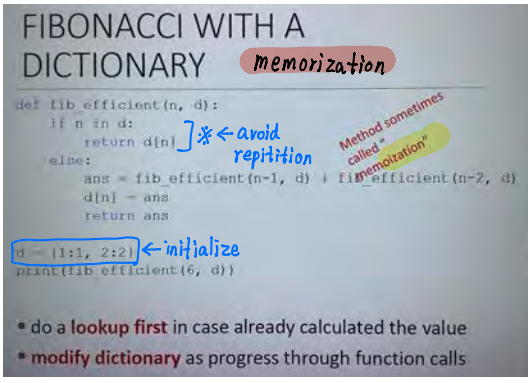
③ Dictionaries – store pairs of data in a compact way

④ dictionary keys must be unique and hashable data types (≈ immutable data)

dictionary values can be any data type;

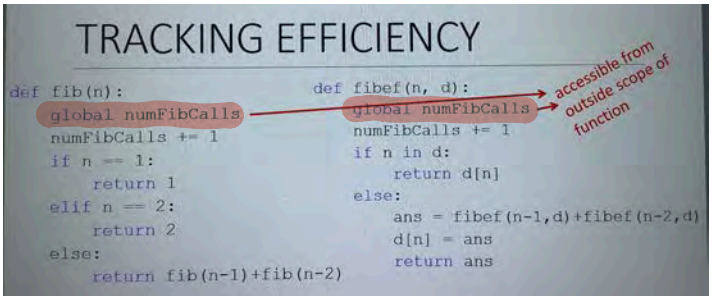
⑤ example: efficient Fibonacci with dictionary (change the Dict while calling)





⑥ Global variable in recursive functions to track total # of function calls (or other grand info)

syntax: > global <varName>



Part Extra – Compact expressions

① Compact expressions for lists syntax: (return a new list)

← optional →

> [<value_to_include> for <element> in <sequence> if <condition>]

e.g.
```
> list1 = [ ]
> for i in range(15):
>     if i % 2 == 0:
>         list1.append(i*3)
```
⇒
```
> [i*3 for i in range(15) if i % 2 == 0]
```

② Compact expressions for dictionaries syntax: (return a new dictionary)

← optional →

> <variable> = { key: new_value for (key, value) in <dictionary>.items( ) if <condition> }

e.g.

```
>>> grades = {"Nora": 90, "Lulu": 15, "Gino": 60}
>>> doubleGrades = {key: value*2 for (key, value) in grades.items() if value % 2 == 0}
>>> doubleGrades
{'Nora': 180, 'Gino': 120}
```

This only includes the item if the original value was even. This is why "Lulu" is not included, because the original value was 15.

Unit 4 – Good programming Practices

Part A – Testing and Debugging

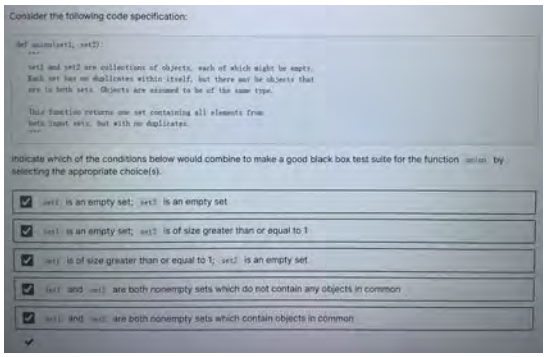
① From start, design code to ease testing and debugging

② During coding process
- break program into modules
- document constraints on modules
- document assumptions

③ make sure having "unit testing" (& "regression testing") before "integration testing"

④ Black-box testing (generally used) vs. Glass-box testing (need to look at the code)

⑤ Tests need to be complete



⑥ Bugs
- Overt and persistent
- Overt and intermittent
- Covert

Easiest to fix ↓ Hardest

⑦ Debugging tools
- built in IDEs
- Python Tutor
- print statement
- use your brain, be systematic in your hunt

"print statement" tool: • good way to test hypothesis

⑧ Error messages: • IndexError • TypeError • NameError • SyntaxError • AttributeError

⑨ Logic Errors – Hard • ZeroDivisionError • ValueError • IOError (e.g. file not found)

Part B – Exceptions and Assertions

① "try-except" method

```
> try:
>     ...
>     ...
> except:
>     ...
```

② Separate except clauses: we can have "except ValueError:" "except ZeroDivisionError:" to execute if specific errors come up

③ Other Exceptions Careful: avoid replacing things in "finally" block!

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)

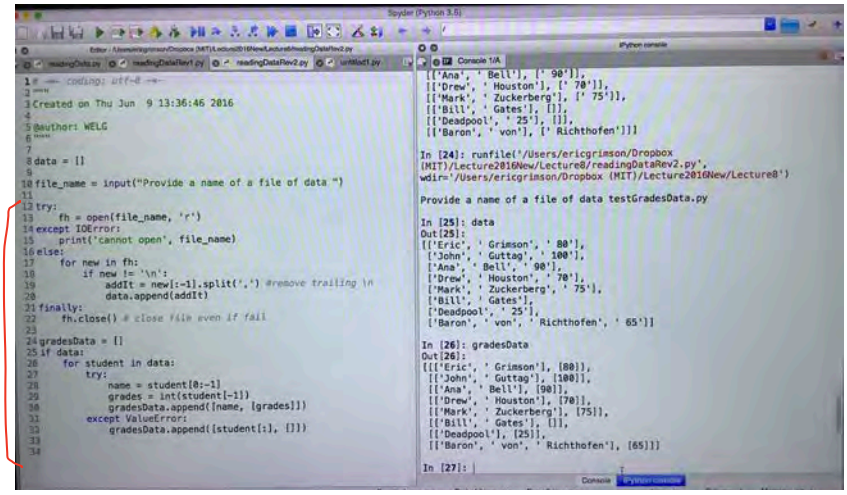
EXAMPLE: CONTROL INPUT

```
data = []
file_name = input("Provide a name of a file of data ")

try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh:
        if new != '\n':
            addIt = new[:-1].split(',') #remove trailing \n
            data.append(addIt)
finally:
    fh.close() # close file even if fail
```

④ "try-except" can use "break", "continue", "return"
↓
jump out of "try-except"

⑤ Example of opening a file & extracting data



⑥ "raise" keyword - exceptions as control flows

raise an exception to check whether "error value" was returned;

⑦ "raise" Syntax: > raise <exceptionName> (<strArguments>)
this will change the "exceptionName" to "strArguments"

⑧ Assertions: used to raise an AssertionError exception if assumptions not met.
(stop immediately)

a supplement to testing
- 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)

Unit 5 – Object Oriented Programming

Part A – Elementary OOP

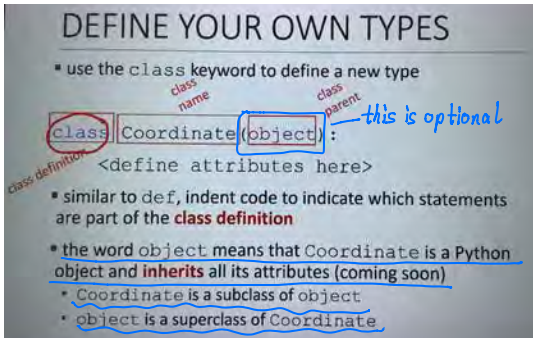
① everything in Python is an object and has a type

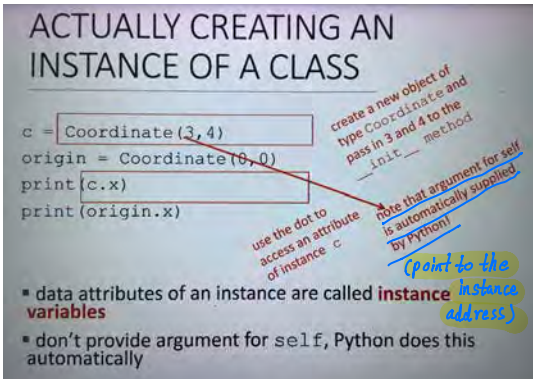
② objects are a data abstraction that capture
- internal representation through attributes
- interface for interacting with object through methods (procedures), defines behaviors but hides implementation.

③ can Destroy objects
- explicitly using "del"
- forget about them

④ Advantages of OOP
- bundle data into packages (abstraction)
- divide-and-conquer development
- classes make it easy to reuse code
  - each class separate environment
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass's behavior

⑤ define own types syntax





⑥ Initializer: > def __init__(self, xx, xx, ...):

Remark: we can define attributes outside "__init__" method.

⑦ Methods within a class: procedural attributes, like a function that works only in this class

> def <methodName>(self, xx, ...):

⑧ Dot Notation: <objectName>.<method>/<attribute>

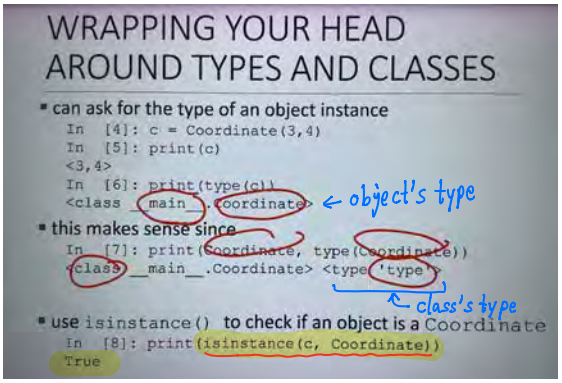
Special: self. refers to "this object"

⑨ Special method names with __<methodName>__(xx, xx, ...):

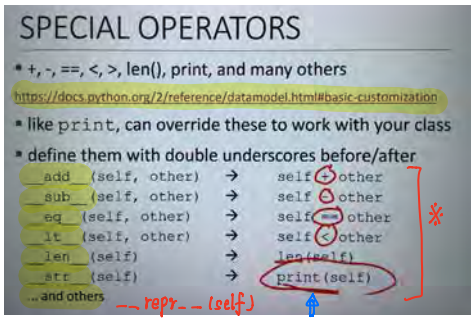
Python approach to operator overloading, allowing classes to define their own behavior with respect to language operators.

⑩ Objects by default return an address. Objects pointing to the same address are same object.

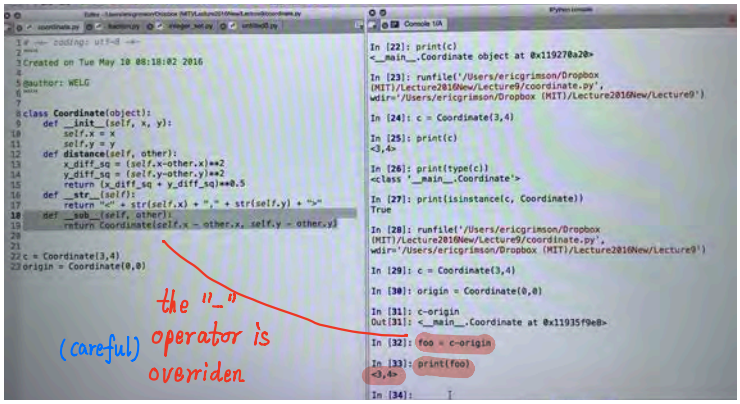
⑪ Types



⑫ Override default operators' methods by __ <methodName> __ (self, xx, ...) inside a class

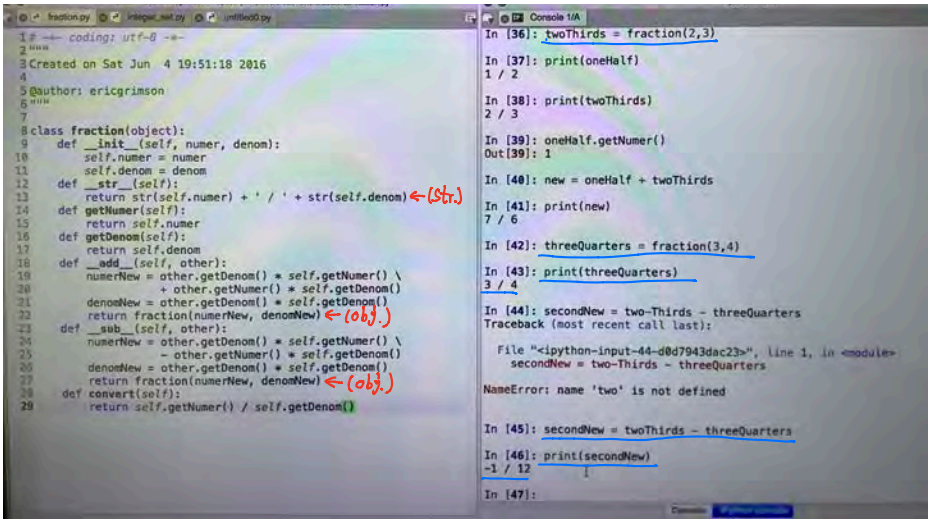


(advanced skill)

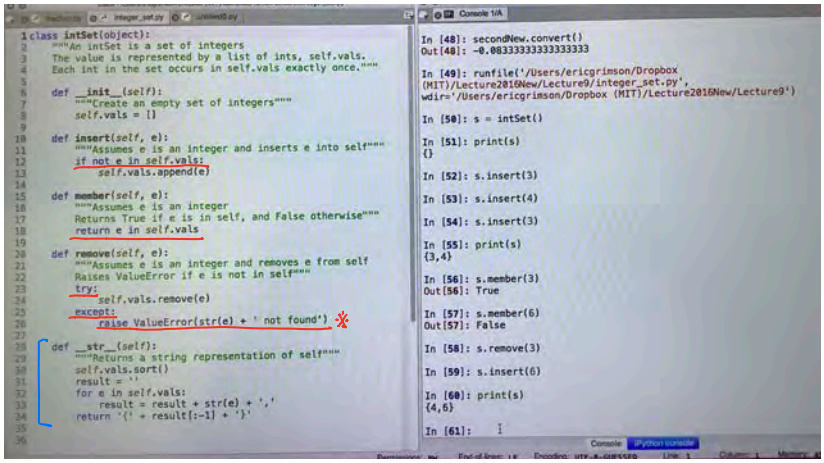


⑬ we should use getters' & setters' methods instead of directly accessing attributes from outside to manipulate & get objects' attributes.

⑭ "Fraction" example with overriding & getters, setters



⑮ Example: Integer Set Class – Utilising "(not) xx in xx" ; raise XX (xx) ; "try-except"
/ "XX (not) in XX"



```
class intSet(object):
    """An intSet is a set of integers
    The value is represented by a list of ints, self.vals.
    Each int in the set occurs in self.vals exactly once."""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}'
```

```
In [48]: secondNew.convert()
Out[48]: -0.08333333333333333

In [49]: runfile('/Users/ericgrimson/Dropbox
(MIT)/Lecture2016New/Lecture9/integer_set.py',
wdir='/Users/ericgrimson/Dropbox (MIT)/Lecture2016New/Lecture9')

In [50]: s = intSet()

In [51]: print(s)
{}

In [52]: s.insert(3)

In [53]: s.insert(4)

In [54]: s.insert(3)

In [55]: print(s)
{3,4}

In [56]: s.member(3)
Out[56]: True

In [57]: s.member(6)
Out[57]: False

In [58]: s.remove(3)

In [59]: s.insert(6)

In [60]: print(s)
{4,6}

In [61]:
```

⑯ eval(<expression>) function : parses the expression passed to this method and runs python expression (code) within the program. ("dynamic" parsing & exe.)

repr(<objName>) function : returns a printable representational string of the given object.
(a string that looks like a valid python expression that could be used to recreate an object with the same value)

⑰ str(<obj.>) / print(<obj.>) vs. repr(<obj.>)
↓ for human beings to read ↓ for programmers & interpreters

remember this

```
>>> # __str__ vs __repr__
>>> import datetime
>>> today = datetime.date.today()
>>>
>>> str(today)
'2017-02-06'
>>> repr(today)
'datetime.date(2017, 2, 6)'
>>> datetime.date(2017, 2, 6)
datetime.date(2017, 2, 6)
>>> today
datetime.date(2017, 2, 6)
>>> # __str__ ==> easy to read, for human consumption
>>> # __repr__ ==> unambiguous
```

```
>>> class Car:
...     def __init__(self, color, mileage):
...         self.color = color
...         self.mileage = mileage
...
...     def __repr__(self):
...         return '{self.__class__.__name__}({self.color}, {self.mileage})'.format(self
=self)
...
...
>>> my_car = Car('red', 37281)
>>> my_car
Car(red, 37281)
>>> str(my_car)
'Car(red, 37281)'
>>> print(my_car)
Car(red, 37281)
>>>
```

rebuilt to avoid users see objects addresses

These 2 are same instruction

Remark: However, >>> str([today, today]) ← in this case, str() functions the same as repr()
'[datetime.date(2021, 3, 22), datetime.date(2021, 3, 22)]'

Remark 2: Best to build "__str__(self, xx, ...)" and "__repr__(self, xx, ...)" for all classes facing users directly. >>> str(<obj>)/print(<obj.>) >>> <objectName>
(users: clients, programmers)

Remark 3: The "__repr__(self, xx, ...)" method Should return a string exactly giving the expression of recreating the same object e.g. Car(red, 37281)
Whenever the user(s) input the object name, it will output expression string instead of address string.

⑱ Pythonic way to have multiple ways of initialization (= multiple constructors in Java)

18.1 default value(s)' parameter(s) in "__init(self,xx,...)__" method
& Call by keyword arguments to call with selected attributes

18.2 Factory methods with @classmethod

```
class Cheese(object):
    def __init__(self, num_holes=0):
        "defaults to a solid cheese"
        self.number_of_holes = num_holes

    @classmethod
    def random(cls):
        return cls(randint(0, 100))

    @classmethod
    def slightly_holey(cls):
        return cls(randint(0, 33))

    @classmethod
    def very_holey(cls):
        return cls(randint(66, 100))

Now create object like this:

gouda = Cheese()
emmentaler = Cheese.random()
leerdammer = Cheese.slightly_holey()
```

18.3 *args & **kwargs
(not recommended)

Now if you want complete freedom of adding more parameters:

```
class Cheese():
    def __init__(self, *args, **kwargs):
        #args -- tuple of anonymous arguments
        #kwargs -- dictionary of named arguments
        self.num_holes = kwargs.get('num_holes',random_holes())
```

To better explain the concept of *args and **kwargs (you can actually change these names):

```
def f(*args, **kwargs):
    print 'args: ', args, ' kwargs: ', kwargs

>>> f('a')
args:  ('a',)  kwargs:  {}
>>> f(ar='a')
args:  ()  kwargs:  {'ar': 'a'}
>>> f(1,2,param=3)
args:  (1, 2)  kwargs:  {'param': 3}
```

http://docs.python.org/reference/expressions.html#calls

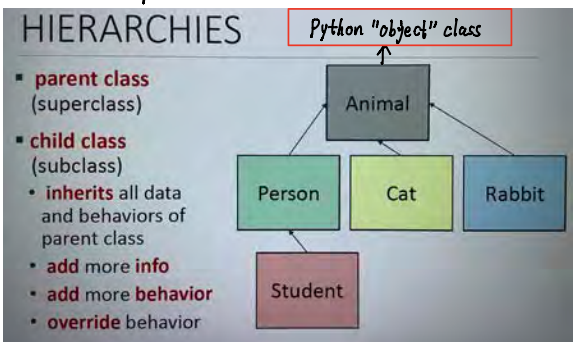
⑲ Implementing the class v.s. Using the class (separate development)

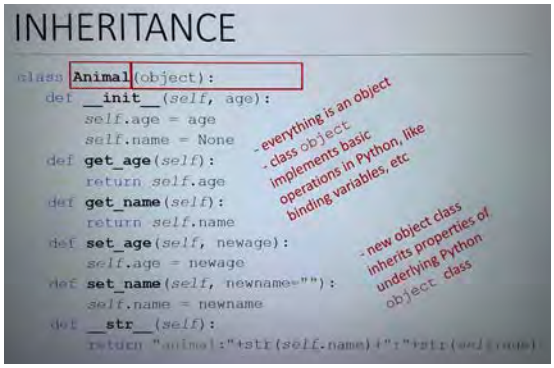
⑳ in Python, better to set a variable's default value "= None" if we do NOT use it temporarily.

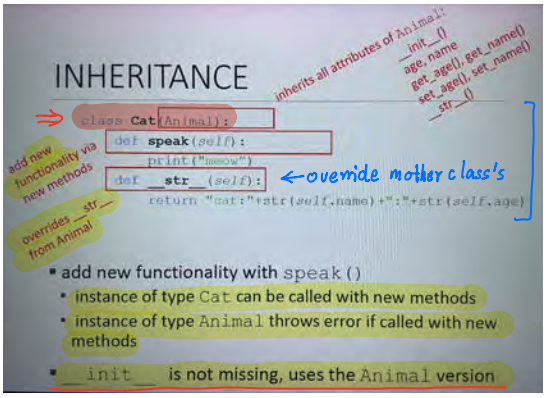
㉑ Python NOT good at information hiding – use Getters & Setters !

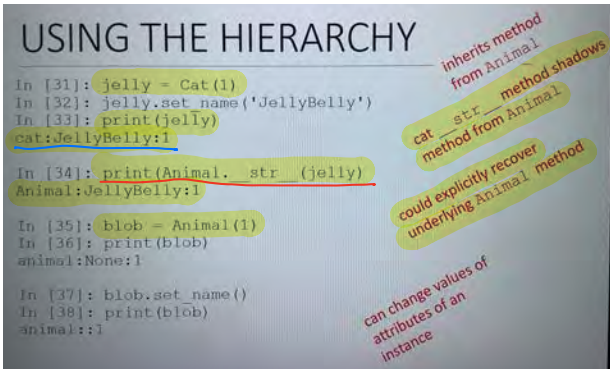
Part B - Hierarchy & Inheritance

① an example to explain basics







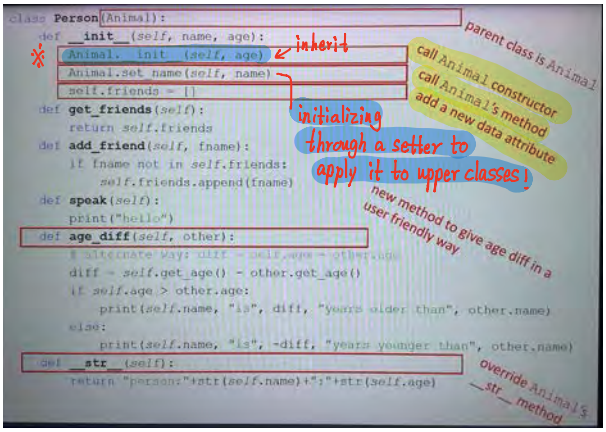


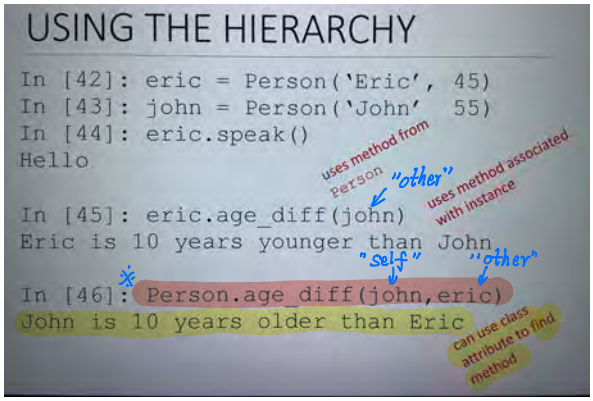
② can use methods from Higher hierarchies, but can NOT use methods for Sub-classes !
(note cannot)

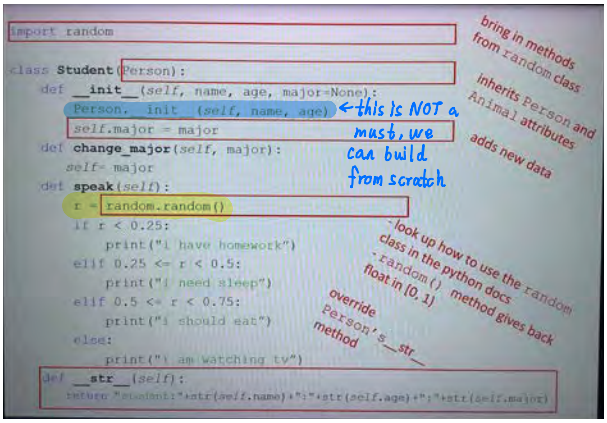
③ Subclass can have methods with same name as other subclasses
(they do NOT disturb each other)

④ sequence for finding a method : (1) in current class → (2) in parent → (3) in grandparent → ...

⑤ Subclasses can have its version initializer, which can be completely new or inherit part of parent's







initializer inheriting same style

⑥ can use "<className>.<methodName>(<selfObj>, <parameter1>, ...)" to directly call the exact method in the exact class

[normal way to call a method : <obj.Name>.<methodName>(<parameter1>, ...)]

※

⑦ whenever we code a subclass, consider "shall we change higher classes' attributes(s) (through setter)?"
i.e. shall we apply something to the whole/other parts of hierarchy ?

⑧ even when calling a superclass's method, the nested method in that method should be found by sequence in ④
※ (starting from the current subclass)

⑨ Logic for finding a method vs. Logic for finding an attribute's value

↓ current class → parent class → ...

same level: left to right

current to highest layer for each super branch

Stop as soon as found (1st found one)

↓ up to bottom (will dig into __init__() methods)

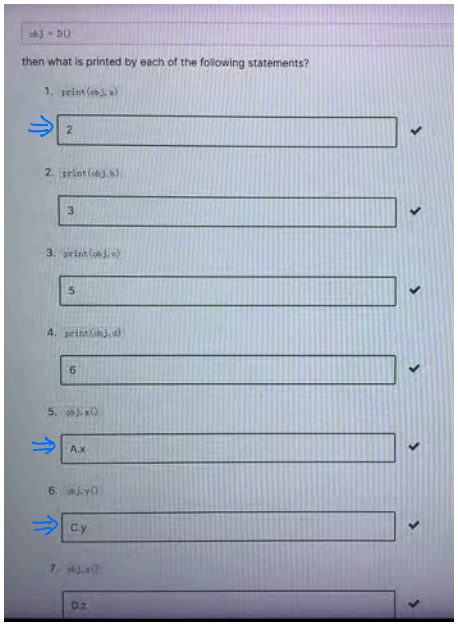
go through ALL in current class & __init__() methods, overwrite former ones & finally assign (super classes')

LAST occurred assignment (the most bottom one)

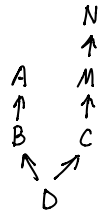
```
class A(object):
    def __init__(self):
        self.a = 1
    def x(self):                    ← Q5
        print("A.x")
    def y(self):
        print("A.y")
    def z(self):
        print("A.z")

class B(A):
    def __init__(self):
        A.__init__(self)
        self.a = 2                  ← Q1
        self.b = 3
    def y(self):
        print("B.y")
    def z(self):
        print("B.z")

class C(object):
    def __init__(self):
        self.a = 4                  ← former one
        self.c = 5
    def y(self):                    ← Q6
        print("C.y")
    def z(self):
        print("C.z")
class D(C, B):                      ← left "C" first in finding a method
    def __init__(self):
        C.__init__(self)
        B.__init__(self)
        self.d = 6
    def z(self):
        print("D.z")
```



⑩ Another illustration of ⑨ finding methods:



Assume "class D(C, B):"

Finding order: D ⇒ C → M → N ⇒ B → A

(stop as soon as found)

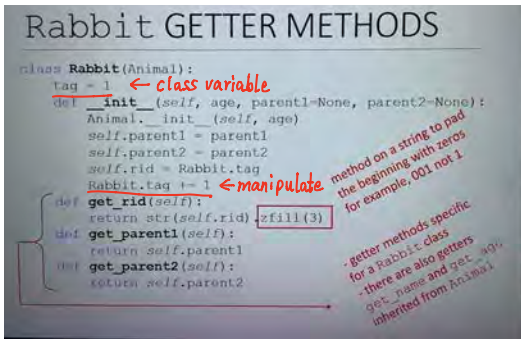
⑪ "up to down codes' sequence" is extremely important – if in the same file, super classes MUST be in front of inheriting subclasses.

⑫ Python do NOT support Java-style function overloading. In each class, among same name methods, ONLY the last one (most bottom one) will function (no error message)

Part C - Class Variables

① Class variables apply to all same class objects (single values)

② 2 ways to creat < "inside __init__(self,...)" method: <className>.<varName> = XX
outside all methods (usually in the begginning): <varName> = XX



Part D - OOP: more concepts & hints

① example of building an application that organizes info about people.

② "__lt__(self,xx)" method will Also overwrite "[<classObj1>, <classObj2>, ...].sort()" default
(of course it will overwrite "<classobj1> < <classobj2>")
The above is due to the fact [xx,xx,...].sort() is built based on "xx < xx"

③ Method shadowing - which class's method is used at runtime? (special "short-circuit" rule from left)

Person class (method: __lt__(self,xx,...))
↑
MITPerson class (method: __lt__(self,xx,...))

Say we create: P1 = Person()
P2 = MITPerson()
Then [P1, P2].sort() will call "Person class" __lt__
But [P2, P1].sort() will call "MITPerson class" __lt__
This may cause error due to diff. attributes of 2 classes. Be careful.

④ Class variables within a class exist in current class's objects & all subclasses' objects (even if parent's __init__ method not called)

⑤ Subclasses can return super classes' methods in their methods to avoid repetition and build linkage
"self" is usually included as a parameter in return methods.

⑥ "isinstance(<objName>, <className>)" is a useful method to see whether obj. in the class

⑦ Substitution Principle in a hierarchy: Shall we insert a new super class inbetween to simplify & organise some of the subclasses?

⑧ classes with nothing but a "pass" is also useful because classes' names themselves are also a kind of information. We can use like "isinstance(<xx>, <xx>)" to check objects' belongings.

⑨ return a class's method parameters vs. call a class's method parameters

↓ must explicitly declare self = ?

↓ "self = <objName>" is automatically assigned

(that's why we call by <objName>.<methodName>(XX))

Part E – "Game: hand" example

① an idiom for counting in a dictionary

> <dictName>[keyName] = <dictName>.get(<keyName>, 0) + 1

If <keyName> not in <dictName>, create & assign 0+1=1; Otherwise +1

② "sorted(<dictName>.keys())" will return a list of keys sorted.

③ "return" statement immediately ends the function

④ when manipulating lists/dictionaries, usually we make a .copy() first to avoid changing the original one.

⑤ Dictionary zero-value keys
- remove: lose historical keys' info
- leave: sometimes cause logic error

two choices have different effects, be careful.

⑥ "try-except" is useful for dictionaries

Part F – "GradeBook" example

① this example create class that includes instances of other classes

② idea: gather together data and procedures for dealing with them in a single structure, so that users can manipulate without having to know internal details.

③ when try to "return" list(s)/dictionary(ies), it's safe to return a copy.

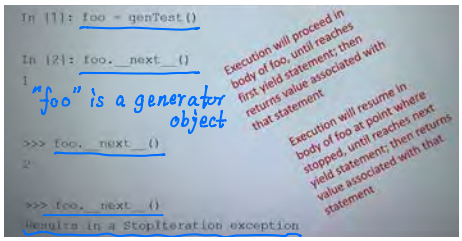
However it's inefficient (imagine big data), we need "Generator" to deal with this issue.

④ Generators: any procedure or method with yield statement (will make the function an object) ※

expression: > yield <any type of data>

"yield <xx>"
- suspends execution and returns a value: "__next__()" starts/resumes execution
- return from a generator raises a StopIteration exception (reach the end of generator)

④' example:
```
> def genTest():
>     yield 1
>     yield 2
```



④ " Fibonacci sequence Generator

```
def genFib():
    fibn_1 = 1 #fib(n-1)
    fibn_2 = 0 #fib(n-2)
※   while True:   ← this is an infinite sequence
        # fib(n) = fib(n-1) + fib(n-2)
        next = fibn_1 + fibn_2
        yield next
        fibn_2 = fibn_1
        fibn_1 = next
```

- evaluating

```
fib = genFib()
```

creates a generator object

- calling

```
fib.__next__()
```

will return the first Fibonacci number, and subsequence calls will generate each number in sequence

- evaluating

```
for n in genFib():
    print(n)
```

will produce all of the Fibonacci numbers (an infinite sequence)

⑤ why Generators

WHY GENERATORS?

- generator separates the concept of computing a very long sequence of objects, from the actual process of computing them explicitly
- allows one to generate each new objects as needed as part of another computation (rather than computing a very long sequence, only to throw most of it away while you do something on an element, then repeating the process)
- have already seen this idea in range

⑥ Fix ③'s inefficiency by Generator

FIX TO GRADES CLASS

Before

```
def allStudents(self):
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    return self.students[:]   ← inefficient
    #return copy of list of students
```

After

```
def allStudents(self):
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    for s in self.students:   ] ← generator
        yield s
```

⑦ "yield" statement will make calling the function gives a "generator" object, instead of returning sth. or None (function calls give "None" by default)

⑧ "Generator" is a typical example to show the power of OOP.

⑨

4.4 Break and Continue Statements, and Else Clauses on Loops

The break statement, like in C, breaks out of the smallest enclosing for or while loop.

The continue statement, also borrowed from C, continues with the next iteration of the loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
```

```
# Note that our solution makes use of the for/else clause, which
# you can read more about here:
# http://docs.python.org/release/1.5/tut/node23.html

def genPrimes():
    primes = []   # primes generated so far
    last = 1      # last number tried
    while True:
        last += 1
        for p in primes:
            if last % p == 0:
                break
        else:
            primes.append(last)
            yield last
```

⑩ we will consider using "Generators" for "infinite things", big data output/utilisation.

Part G – Example: Encryption by Caesar cipher method

① Caesar cipher is NOT very secure: shift each letter to +ith letter in the alphabet.

② Open, Read and Deal with .txt files

```python
import string

### DO NOT MODIFY THIS FUNCTION ###
def load_words(file_name):
    '''
    file_name (string): the name of the file containing
    the list of words to load

    Returns: a list of valid words. Words are strings of lowercase letters.

    Depending on the size of the word list, this function may
    take a while to finish.
    '''
    print('Loading word list from file...')
    # inFile: file
    in_file = open(file_name, 'r')
    # line: string
    line = in_file.readline()
    # word_list: list of strings
    word_list = line.split()
    print('  ', len(word_list), 'words loaded.')
    in_file.close()
    return word_list
```

"read" mode

ONLY useful for small data ("for line in in_file" is better)

```python
### DO NOT MODIFY THIS FUNCTION ###
def is_word(word_list, word):
    '''
    Determines if word is a valid word, ignoring
    capitalization and punctuation

    word_list (list): list of words in the dictionary.
    word (string): a possible word.

    Returns: True if word is in word_list, False otherwise

    Example:
    >>> is_word(word_list, 'bat') returns
    True
    >>> is_word(word_list, 'asdf') returns
    False
    '''
    word = word.lower()
    word = word.strip(" !@#$%^&*()-_+={}[]|\:;'<>?,./\"")
    return word in word_list

### DO NOT MODIFY THIS FUNCTION ###
def get_story_string():
    """
    Returns: a joke in encrypted text.
    """
    f = open("story.txt", "r")
    story = str(f.read())
    f.close()
    return story
```

for small data

③ for "Getter" method(s), if return mutable data, return a copy !!

④ String is in nature a list

⑤ "import string" class :
- 5.1 > string.ascii_lowercase — a string "ab…yz"
- 5.2 > string.ascii_uppercase — a string "AB…YZ"
- 5.3 > string.punctuation — a string "!"#$%…{|}~"
- 5.4 > string.digits — a string "0123456789"

⑥ if in dict.keys(), return value; else return itself (unchange)

```
> for char in text:
>     <listName>.append(<dictName>.get(char,char))
```

default: itself (if NOT in .keys())

⑦ other than "__init__()" inheritance, better use self.<methodName>(xx,...) to call all methods within the hierarchy.

In fact, we can use > super().__init__(xx,...) to replace

> <superclassName>.__init__(self,xx,...)

⑧ Although "None" is a good initial value, we should assign specific type data to avoid error if we already know the variable's type. "None" is only used when we can not decide the variable's type yet.

Unit 6 – Algorithmic Complexity

Part A – Computational Complexity

① usually focus on time efficiency

② How to evaluate Efficiency { with a timer?
count the operations?
abstract notion of order of growth? ✓ most appropriate way

③ use a timer

"import datetime" class: 3.1 > datetime.datetime.now() time accurate to seconds
3.2 > datetime.date.today() today's date

"import time" class: 3.3 > time.time() time converted to a large float

④ Orders of Growth as inputs increase – a rough but useful measurement.

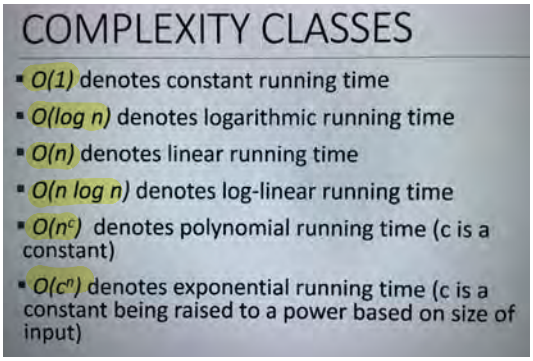
⑤ We care about worst cases (Worst cases can be those run till the end.)
this is a "tight bound" (not upper bound because we can set unreasonable high upper bound)

⑥ Complexity classes

Loops & Function calls (including recursive calls) matter a lot

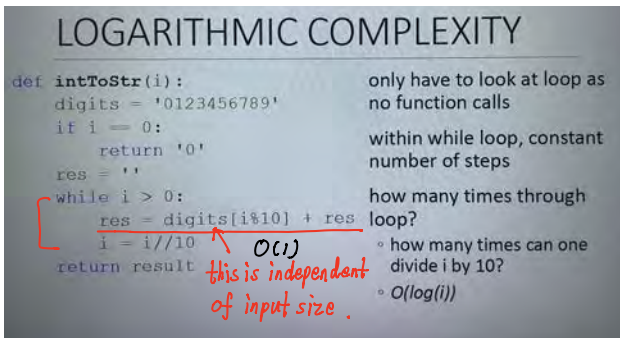
(worst cases – "break" changes nothing)

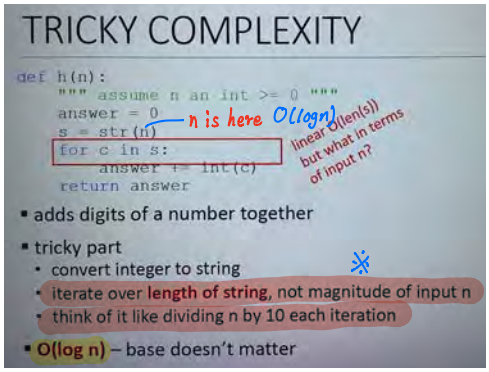
Remark: Complexity MUST be in terms of inputs



⑦ Constant Complexity: can have loops or recursive calls, but # of iterations or calls independent of size of input.

⑧ Logarithmic Complexity: complexity grows as log of size of one of its inputs.
e.g. bisection search; binary search of a list





⑨ Linear Complexity : • Search a list in sequence to see if an element is present

• add characters of a string • 1 recursive call at a time

```
def fact_recur(n):
    """ assume n >= 0 """
    if n <= 1:
        return 1
    else:
        return n*fact_recur(n - 1)
```

only 1 recursive call

- computes factorial recursively
- if you time it, may notice that it runs a bit slower than iterative version due to function calls
- still O(n) because the number of function calls is linear in n
- iterative and recursive factorial implementations are the same order of growth

COMPLEXITY OF ITERATIVE FIBONACCI

```
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

constant O(1) constant O(1) linear O(n) constant O(1)

- Best case: O(1)
- Worst case: O(1) + O(n) + O(1) → O(n)

⑩ Log-Linear Complexity : Fastest Sort - Merge sort $O(N(\log N))$

⑪ Polynomial Complexity : find the "most" nested loops, say nesting c times, then big-O is $O(N^c)$ (careful about the base N variation)

QUADRATIC COMPLEXITY

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

first nested loop takes *len(L1)*len(L2)* steps

second loop takes at most *len(L1)* steps

latter term overwhelmed by former term

*O(len(L1)*len(L2))*

Remark: immutable data concatenation repeated by may result in quadratic complexity

(but must be directly related to input size)

⑫ Exponential Complexity (expensive) : e.g. Towers of Hanoi , ≥ 2 recursive calls at the same time

EXPONENTIAL COMPLEXITY

```
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

← return all subsets of L

but important to think about size of smaller

know that for a set of size k there are 2^k cases

so to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps

math tells us this is $O(2^n)$

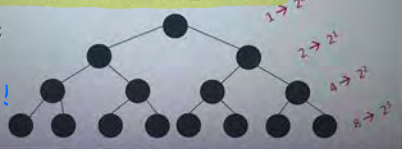
COMPLEXITY OF RECURSIVE FIBONACCI

```
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

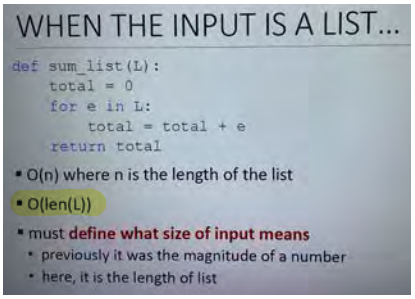
2 recursive calls at same time

- Worst case: $O(2^n)$

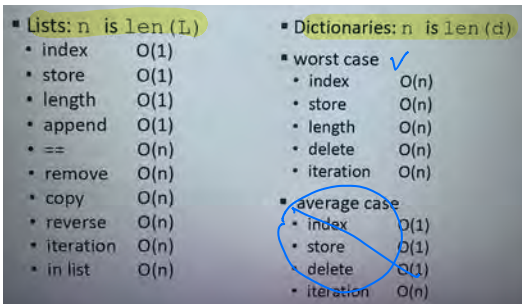
this is bad!



⑬ when the input is a List



⑭ Complexity of common Python Function



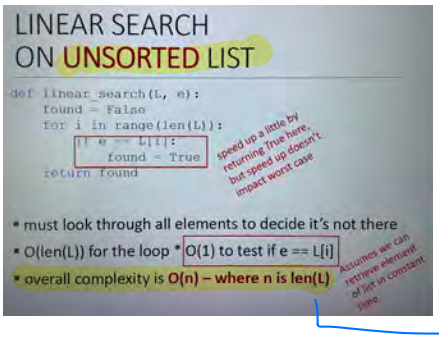
⑮ Normally, the minimum operations (≈ complexity) is the total # of steps / calls / prints.

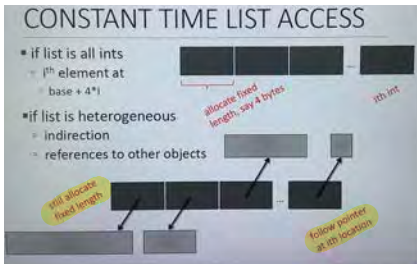
⑯ copying & pass as parameter may significantly increase complexity
(but we often need copying for mutable data)
(we can choose NOT to copy but must ensure no mutable parameters modification for all subfunctions)

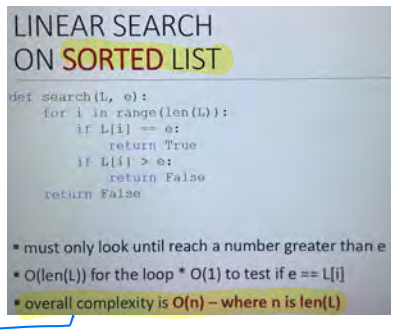
Part B – Searching and Sorting Algorithms – Searching

① early exist by boolean indicator usually does NOT affect complexity.

② Linear Search : brute force search ; Lists does NOT have to be sorted



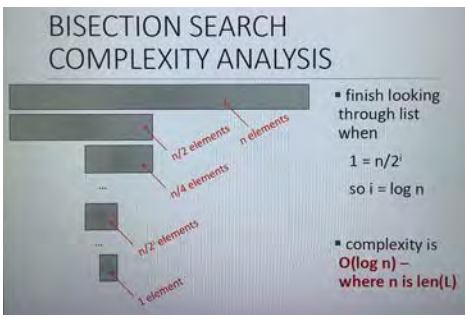


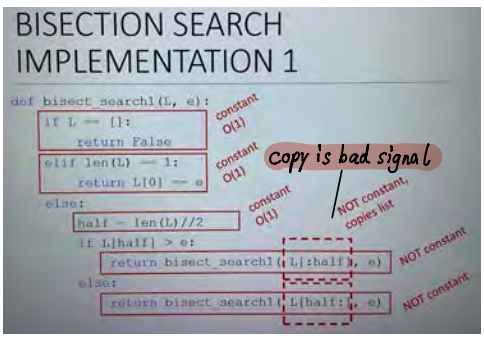


Sorting does NOT change complexity in "linear search"

③ Bisection Search : divide-and-conquer algorithm

※ Lists MUST be Sorted to give correct answer





BISECTION SEARCH IMPLEMENTATION 2 O(log n)

※this version is chosen

```
def bisect_search2(L, e):
    def bisect_search_helper(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bisect_search_helper(L, e, low, mid - 1)
        else:
            return bisect_search_helper(L, e, mid + 1, high)
    if len(L) == 0:
        return False
    else:
        return bisect_search_helper(L, e, 0, len(L) - 1)
```

NOT constant

NOT constant

indices

COMPLEXITY OF THE TWO BISECTION SEARCHES

- **Implementation 1 – bisect_search1**
  - O(log n) bisection search calls
  - O(n) for each bisection search call to copy list
  - → **O(n log n)**
  - → O(n) for a tighter bound because length of list is halved each recursive call
- **Implementation 2 – bisect_search2** and its helper
  - pass list and indices as parameters
  - ※ list never copied, just re-passed
  - → **O(log n)**

④ Why bother Sorting first for searching? (Lists only)

Because in many cases, we sort a list once then do many searches and SORT + k* O(log n) < k*O(n) is always true for large k! (amortised cost lower)

Part C – Searching and Sorting Algorithms – Sorting

① Monkey Sort / Bogo Sort worst cases – unbound

② Bubble Sort : compare consecutive pairs of elements, swap & start over again until all finish.

O(n²)

COMPLEXITY OF BUBBLE SORT

```
def bubble_sort(L):
    swap = False
    while not swap:                    O(len(L))
        swap = True
        for j in range(1, len(L)):     O(len(L))
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **O(n²) where n is len(L)**
  to do len(L)-1 comparisons and len(L)-1 passes

③ Selection Sort : constantly keep the left portion of the list sorted

O(n²)

SELECTION SORT

- first step
  - extract **minimum element**
  - **swap it** with element at **index 0**
- subsequent step
  - in remaining sublist, extract **minimum element**
  - **swap it** with the element at **index 1**
- keep the left portion of the list sorted
  - at ith step, **first i elements in list are sorted**
  - all other elements are bigger than first i elements

COMPLEXITY OF SELECTION SORT

```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):                    len(L) times → O(len(L))
        for i in range(suffixSt, len(L)):        len(L) - suffixSt times → O(len(L))
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1
```

2 layers loops

pythonic swap

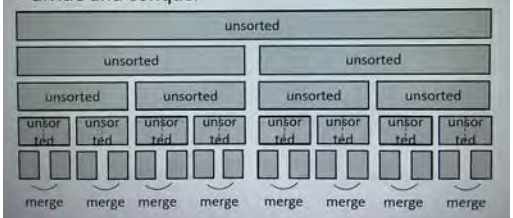
- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **O(n²) where n is len(L)**

④ Merge Sort : divide-and-conquer approach → split into 2 part and then merge (recursively)

O(n(log n))

MERGE SORT

- divide and conquer



- split list in half until have sublists of only 1 element

EXAMPLE OF MERGING

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1, 2	[]
[5,12,18,19,20]	[2,3,4,17]	5, 2	[1]
[5,12,18,19,20]	[3,4,17]	5, 3	[1,2]
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

MERGE SORT ALGORITHM -- RECURSIVE

O(log n)

```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

base case

divide

conquer with the merge step

- divide list successively into halves
- depth-first such that conquer smallest pieces down one branch first before moving to larger pieces

MERGING SUBLISTS STEP

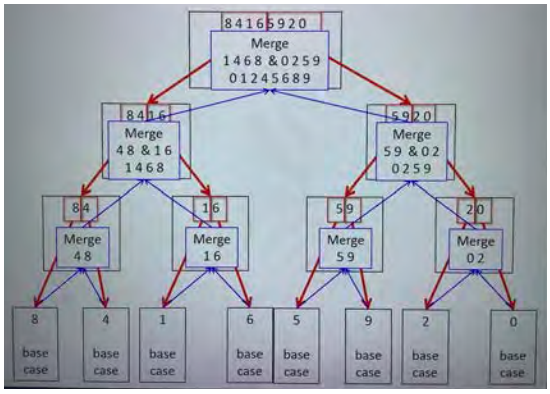
O(n)

```
def merge(left, right):
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

- left and right sublists are ordered
- move indices for sublists depending on which sublist holds next smallest element

when right sublist is empty

when left sublist is empty



COMPLEXITY OF MERGE SORT

- at first recursion level
  - n/2 elements in each list
  - O(n) + O(n) = O(n) where n is len(L)
- at second recursion level
  - n/4 elements in each list
  - two merges → O(n) where n is len(L)

focus on occurring / simultaneously

- each recursion level is O(n) where n is len(L)
- dividing list in half with each recursive call
  - O(log(n)) where n is len(L)
- overall complexity is O(n log(n)) where n is len(L)

Unit 7 – Plotting

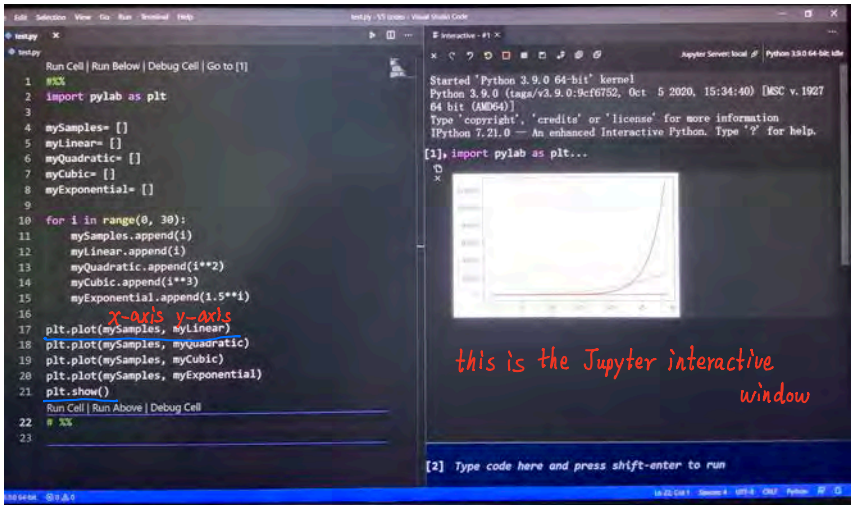
Part A – Visualizing Results: Pylab (from "matplotlib")

⓪ MUST install "numpy" & "matplotlib" first

① "import pylab" to start visualization: it provides access to existing set of graphing/plotting procedures

② For more details, go to see "pylab" documentation

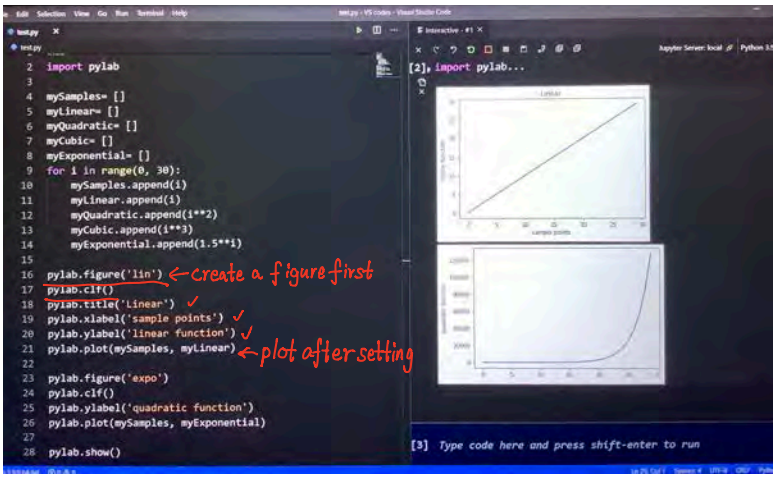
③ "pylab.plot(<1stname>, <2ndname>)" plot two lists (shown below)
↑ x-axis ↑ y-axis
"pylab.show()" show a separate window of plotting result



④ "pylab.figure(<graphName>)"
↑ create a separate figure (usually code this line first)

"pylab.xlabel(<labelName>)" "pylab.ylabel(<labelName>)"
"pylab.title(<titleName>)"
Set texts for each functionality described

clean up previous created windows (avoid figure formatting error) "pylab.clf()"
↑ usually after ".figure(xx)"



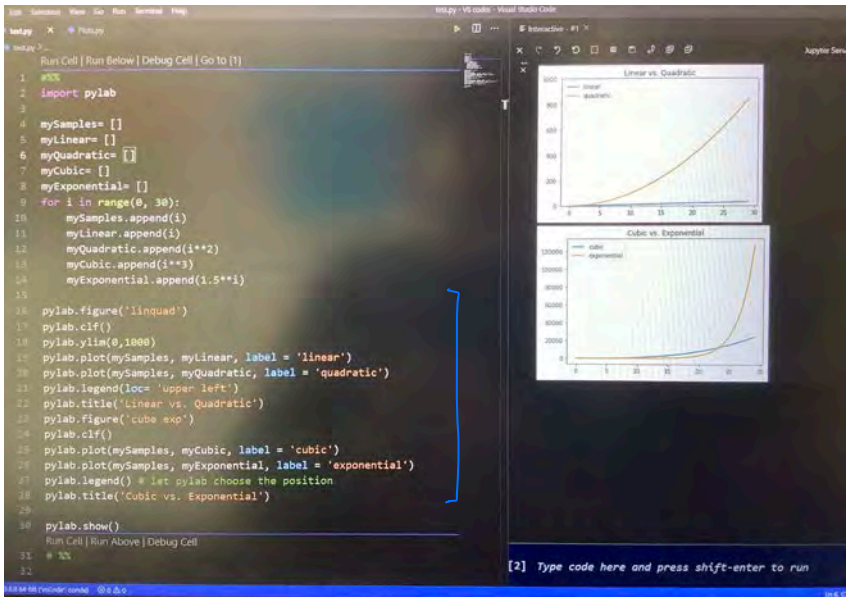
⑤ Comparing Plots:

5.1 changing Limits on axes — pylab.xlim(<minNum>, <maxNum>); pylab.ylim(<minNum>, <maxNum>) (before ".plot(xx)" function)

5.2 overlaying plots — write ".plot(a,b) & .plot(c,d) & ..." under same ".figure(< >)"

5.3 Adding more parameter(s) — pylab.plot(<list1>, <list2>, label = <strName>)

5.4 set a location for 5.3 label(s) — "pylab.legend(loc = <locationArgument>)" (optional; without it, pylab will choose the position)

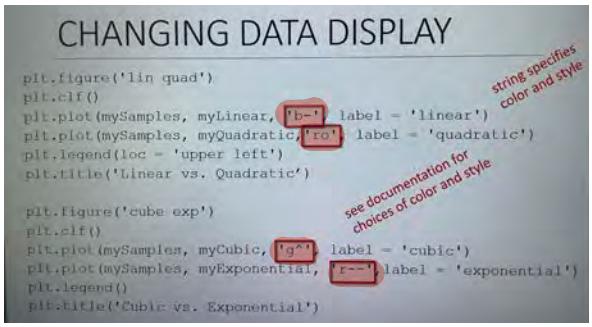


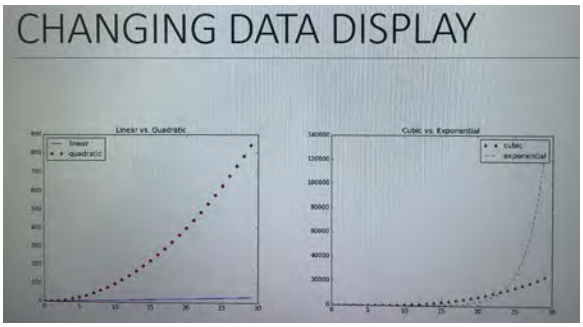
⑥ Control display parameters

6.1 Colors & Line types — parameters in ".plot(<list1>, <list2>, 'ox', label = <strName>)" function ('ox' is 3rd argument (optional))

'o' in 'ox' stands for colors — b: blue; r: red; g: green; k: black

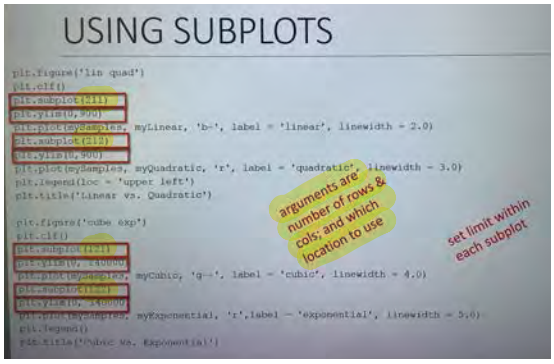
'x' in 'ox' stands for line types — -: normal line; o: dot line; ^: triangle line; --: dash line

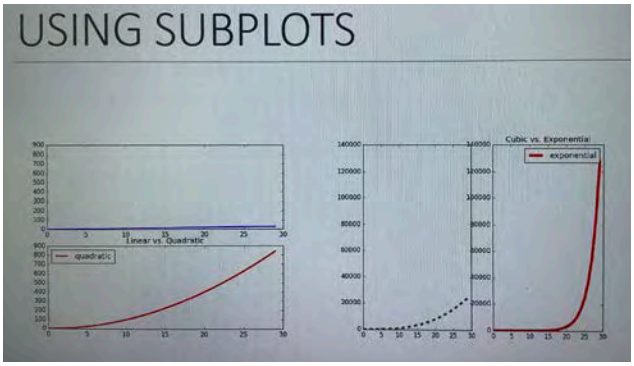




6.2 Line width (5th parameter, 线的粗细) in ".plot(<list1>, <list2>, 'ox', label=<>, linewidth = <float>)"

6.3 Using Subplots "pylab.subplot('OXT')" 'O': # of rows; 'X': # of cols; 'T': location

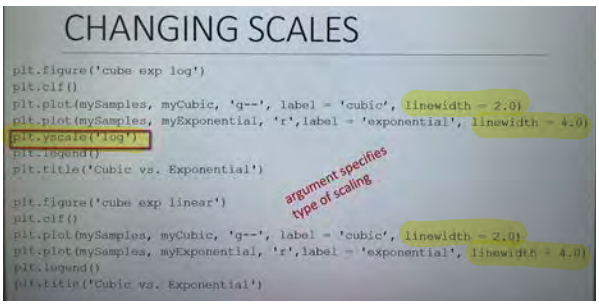


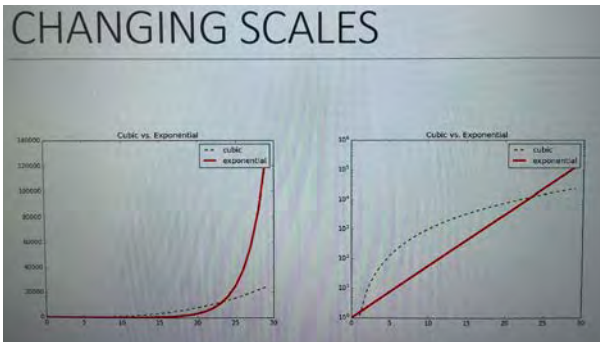


6.4 Changing Scales "pylab.xscale(<argument>)" "pylab.yscale(<argument>)"

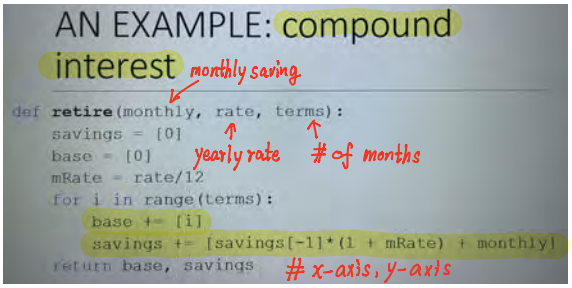
e.g. "log" type of scaling

e.g. pylab.yscale('log') will change y-axis to "logarithm" scale, which makes an exponential linear

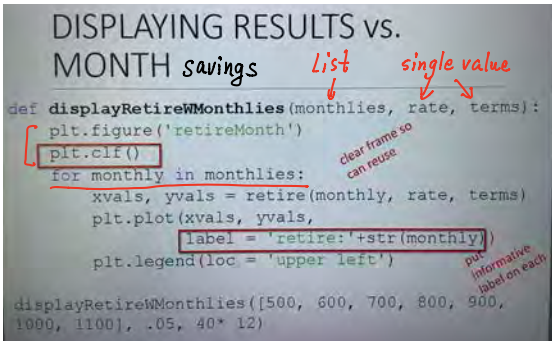


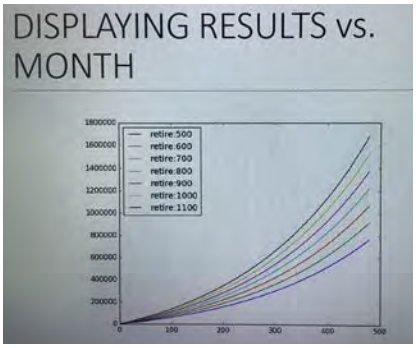


Part B - Retirement Example (Compound interest)



① vs. monthly savings (input: list-monthlies)





② vs. yearly rates (input: list - rates)

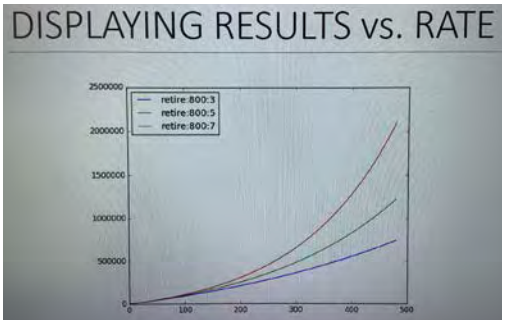
DISPLAYING RESULTS vs. RATE

```
def displayRetireWRates(month, rates, terms):
    plt.figure('retireRate')
    plt.clf()
    for rate in rates:
        xvals, yvals = retire(month, rate, terms)
        plt.plot(xvals, yvals,
                 label = 'retire:'+str(month)+ ':' + \
                         str(int(rate*100)))
    plt.legend(loc = 'upper left')

displayRetireWRates(800,[.03, .05, .07], 40*12)
```

list

put informative label on each



③ vs. monthly savings + yearly rates (input: lists - monthlies & rates) (2-layers loop)

DISPLAYING RESULTS vs. BOTH

```
def displayRetireWMonthsAndRates(monthlies, rates, terms):
    plt.figure('retireBoth')
    plt.clf()
    plt.xlim(30*12, 40*12)
*   monthLabels = ['r', 'b', 'g', 'k']
    rateLabels = ['-', 'o', '--']
    for i in range(len(monthlies)):
        monthly = monthlies[i]
        monthLabel = monthLabels[i%len(monthLabels)]
        for j in range(len(rates)):
            rate = rates[j]
            rateLabel = rateLabels[j%len(rateLabels)]
            xvals, yvals = retire(monthly, rate, terms)
            plt.plot(xvals, yvals,
                     monthLabel+rateLabel,
                     label = 'retire:'+str(monthly)+ ':' \
                             + str(int(rate*100)))
    plt.legend(loc = 'upper left')

displayRetireWMonthsAndRates([500, 700, 900, 1100], [.03, .05, .07],
                             40*12)
```

create sets of labels

pick new label for each month choice

pick new label for each rate choice

create label for plot