

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
 - < **syntactic 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 interpreter in a shell**
 - (commands (statements) instruct interpreter 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 $\ggg 4 > 5$ or $3 < 4$ and $9 > 8$ (**True**)

Operator Precedence

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

(12) $\ggg 5/2 == 5/2.0$ (True) $\ggg 5*2 == 5.0*2.0$ (True)
 $\ggg 2**2 == 2.0**4$ (False) Tricky Comparisons

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

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

(14) String multiplication:

e.g. $\ggg "bc" * 3$
 $\quad \quad \quad "bcbcbc"$

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

$[-1]$ $[a:b]$ $[a:]$ $[:a]$ $[:]$ all acceptable
 end indices NOT included

(15) Tricks of "print(XXX)"

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

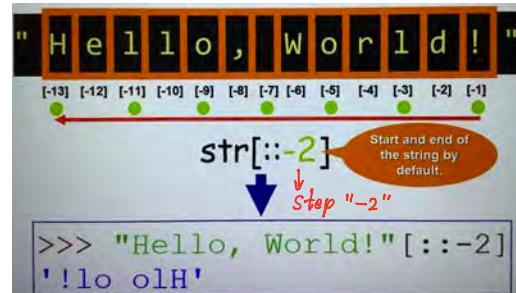
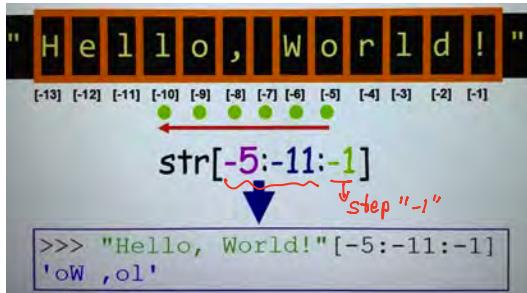
```
In [80]: print("my fav num is", x, ".", "x =", x)
my fav num is 1 . x = 1
          ^ comma in print() default is a space

In [81]: print("my fav num is" + x_str + "," + "x = " + x_str)
my fav num is1.x = 1
          ^ "+" no default

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

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

$\ggg str[4] = "elloworld"$
 $\ggg str[4[1:9:2]]$ "elur" range [1,9], every other char (step:+2) ∴ 1,3,5,7 indices
 $\ggg str[4[:::-1]]$ "dlrowolleh" range all, inverse order (note str[::1] NOT work)



(17) do NOT name a variable "sum"

(18) Tricky Q

```
1. num = 10
for num in range(5):
    print(num)
    print(num)

0, 1, 2, 3, 4, 4

2. divisor = 2
for num in range(0, 10, 2):
    print(num/divisor)

"/" in Python returns float
0.0, 1.0, 2.0, 3.0, 4.0
```

(19) Casting: e.g. $> a = int('4')$

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

in Python, the interpreter 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

②) 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) != 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
print('The binary representation of the decimal ' + str(x) + ' is ' + result)

```

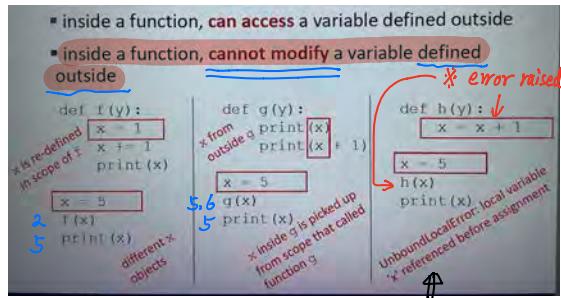
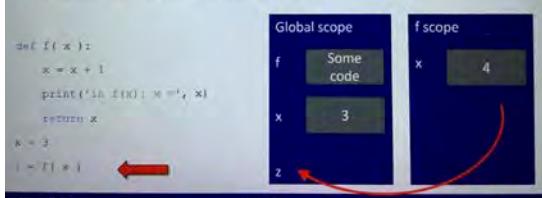
Total length
 fill "0" in front fraction positions
 if necessary

Unit 2 – Simple Programs

Part A - Normal Functions

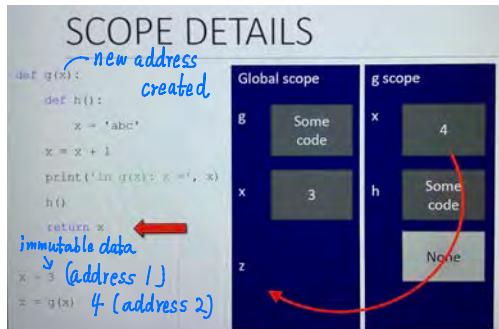
① Variable Scope

- **scope** is mapping of names to objects

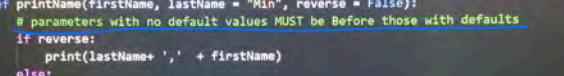


No error if pass
by parameter!!

- 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

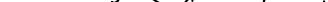


The screenshot shows a Python script named `test.py` in VS Code. The code defines a function `printName` that prints two arguments. It includes examples of calling the function with keyword arguments and without them.

```
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
```

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

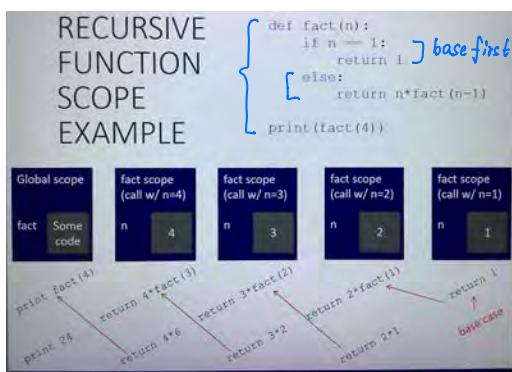
- ⑥ Doc String in each Function (not mandatory, but Recommended) 

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.



SOME OBSERVATIONS

- If mutable data, MUST pass a copy *
- each recursive call to a function creates its own scope/environment
- bindings of variables in a scope is not changed by recursive call (immutable data)
- flow of control passes back to previous scope once function call returns value

* using the same variable names but they are different objects in separate scopes

- ⑤ Inductive Reasoning - ensure calling itself with a smaller parameter(s) scope
 - 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>)

FILES: example

```

nameHandle = open('kids', 'w')
for i in range(2):
    name = input('Enter name: ')
    nameHandle.write(name + '\n')
nameHandle.close()
  
```

FILES: example

```

nameHandle = open('kids', 'r')
for line in nameHandle:
    print(line)
nameHandle.close()
  
```

Unit 3 - Structured Types

Part A - Tuples

- ① Tuples can be added "+", and sliced "[a:b]", but can NOT be modified
- ② Non-empty tuples MUST include at least ONE comma "," (NOT change address/value)
- 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 elements(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 Conversion with Strings
 - string to list with every character an element: > list(s) *
 - split a string to list with separator: > 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)
- ⑤ Function calls vs. OO methods
 - e.g. sorted(L) → returns a new sorted copy, NOT mutate L
 - ↑ Create a new copy ↑ modify the original one
 - 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 Lists(s)

MUTATION AND ITERATION

* avoid mutating a list as you are iterating over it

```
def remove_dups(L1, L2):  
    for e in L1:  
        if e in L2:  
            L1.remove(e)
```

X

```
L1 = [1, 2, 3, 4]  
L2 = [1, 2, 5, 6]  
remove_dups(L1, L2)
```

✓

```
def remove_dups_new(L1, L2):  
    L1_copy = [L1[i] for i in range(len(L1))]  
    for e in L1_copy:  
        if e in L2:  
            L1.remove(e)
```

clone list first note
that L1_copy ->
does NOT clone

* L1 is [2, 3, 4] not [3, 4] Why?
• Python uses an internal counter to keep track of index it is in the loop
• mutating changes the list length but Python doesn't update the counter
• loop never sees element 2

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.

GENERALIZATION OF HOPS

```
# Python provides a general purpose HOP, map
# simple form - a unary function and a collection of suitable arguments
map(abs, [1, -2, 3, -4])
# produces an "iterable", so need to walk down it
for elt in map(abs, [1, -2, 3, -4]):
    print(elt)
# remember range?
[1, 2, 3, 4]

# general form - an n-ary function and n collections of arguments
L1 = [1, 28, 36]
L2 = [2, 57, 9]
for elt in map(min, L1, L2):
    print(elt)
# Compare each element
# & return "each" min element!
[1, 28, 9]
```

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 (\approx immutable data)

dictionary values can be any data type ;

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

INEFFICIENT FIBONACCI

```
fib(n) = fib(n-1) + fib(n-2)
```

- recalculating the same values many times!
- could keep track of already calculated values

FIBONACCI WITH A DICTIONARY memorization

```
def fib_efficient(n, d):
    if n in d:
        return d[n] #<-- avoid repetition
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
    return ans

d = {1:1, 2:2} #<-- initialize
print(fib_efficient(6, d))
```

- do a lookup first in case already calculated the value
- modify dictionary as progress through function calls

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

syntax: `> global <varName>`

TRACKING EFFICIENCY

```
def fib(n):
    global numFibCalls
    numFibCalls += 1
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1)+fib(n-2)

def fibef(n, d):
    global numFibCalls
    numFibCalls += 1
    if n in d:
        return d[n]
    else:
        ans = fibef(n-1,d)+fibef(n-2,d)
        d[n] = ans
    return ans
```

A red arrow points from the 'global numFibCalls' declaration in the first function to the second, with the text 'accessible from outside scope of function'.

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

Consider the following code specification:

```
def union(sset1, sset2):
    ...
    sset1 and sset2 are collections of objects, each of which might be empty.
    Each set has no duplicates within itself, but there may be objects that
    are in both sets. Objects are assumed to be of the same type.
    This function returns one set containing all elements from
    both input sets, but with no duplicates.
    ...
    
```

Indicate which of the conditions below would combine to make a good black box test suite for the function `union`, by selecting the appropriate choice(s).

<input checked="" type="checkbox"/> <code>sset1</code> is an empty set; <code>sset2</code> is an empty set
<input checked="" type="checkbox"/> <code>sset1</code> is an empty set; <code>sset2</code> is of size greater than or equal to 1
<input checked="" type="checkbox"/> <code>sset1</code> is of size greater than or equal to 1; <code>sset2</code> is an empty set
<input checked="" type="checkbox"/> <code>sset1</code> and <code>sset2</code> are both nonempty sets which do not contain any objects in common
<input checked="" type="checkbox"/> <code>sset1</code> and <code>sset2</code> are both nonempty sets which contain objects in common

- ⑥ Bugs
 - Overt and persistent
 - Overt and intermittent
 - Covert
 - ⑦ Debugging tools
 - built in IDEs
 - Python Tutor
 - print statement
- Easiest to fix
- ↓
- Hardest
- "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

The screenshot shows the Spyder Python 3.5 IDE interface. On the left, the code editor contains a script named `readingDataRev2.py` with the following content:

```
1<-- coding: utf-8 --
2
3Created on Thu Jun 9 13:36:46 2016
4
5@authors: WELG
6
7
8data = []
9
10file_name = input("Provide a name of a file of data ")
11
12try:
13    fh = open(file_name, 'r')
14except IOError:
15    print('cannot open', file_name)
16else:
17    for new in fh:
18        if new != '\n':
19            addIt = new[:-1].split(',') #remove trailing \n
20            data.append(addIt)
21    finally:
22        fh.close() # close file even if fail
23
24gradesData = []
25if data:
26    for student in data:
27        try:
28            name = student[0:-1]
29            grades = student[-1]
30            gradeData.append([name, [grades]])
31        except ValueError:
32            gradesData.append([student[1], []])
33
34
```

On the right, the IPython console shows the execution of the code and the resulting data:

```
In [24]: runfile('/Users/ericgrimson/Dropbox (MIT)/Lecture08/New/Lecture8/readingDataRev2.py', wdir='/Users/ericgrimson/Dropbox (MIT)/Lecture08/New/Lecture8')
Provide a name of a file of data testGradesData.py

In [25]: data
Out[25]:
[['Eric', 'Grimson', '88'],
 ['John', 'Guttag', '100'],
 ['Ana', 'Bell', '98'],
 ['Drew', 'Houston', '70'],
 ['Mark', 'Zuckerberg', '75'],
 ['Bill', 'Gates',
  ['Deadpool', '25'],
 ['Baron', 'von', 'Richthofen', '85']]
```

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

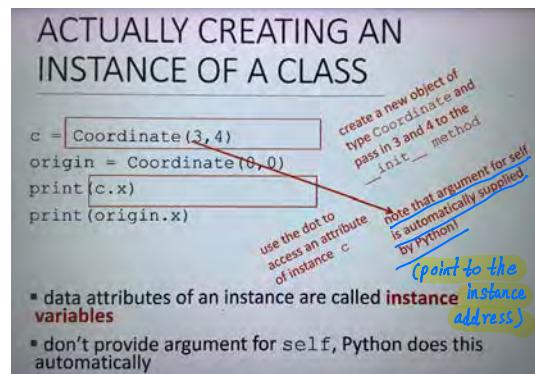
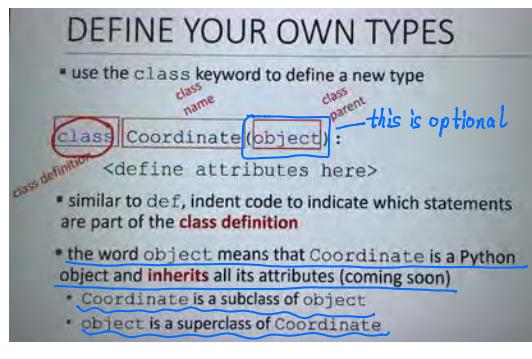
check for violations of constraints on procedure

(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: <code><objectName>. <method>/<attribute></code>

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

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- * can ask for the type of an object instance


```
In [4]: c = Coordinate(3,4)
In [5]: print(c)
<3, 4>
In [6]: print(type(c))
<class '__main__.Coordinate'>
```

object's type
- * this makes sense since


```
In [7]: print(Coordinate, type(Coordinate))
<class '__main__.Coordinate'> <type 'type'>
```

class's type
- * use isinstance() to check if an object is a Coordinate


```
In [8]: print(isinstance(c, Coordinate))
True
```

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

SPECIAL OPERATORS

- * +, -, ==, <, >, len(), print, and many others
- <https://docs.python.org/2/reference/datamodel.html#basic-customization>
- * like print, can override these to work with your class
- * define them with double underscores before/after

<code>__add__(self, other) → self + other</code>	<code>__sub__(self, other) → self - other</code>
<code>__eq__(self, other) → self == other</code>	<code>__lt__(self, other) → self < other</code>
<code>__len__(self) → len(self)</code>	<code>__str__(self) → print(self)</code>
...and others	<code>__repr__(self)</code>

*

(advanced skill)

```

In [22]: print(c)
<__main__.Coordinate object at 0x119278a20>

In [23]: runfile('/Users/ericgrimson/Dropbox/MIT/Lecture2010New/Lecture9/coordinate.py',
              wdir='/Users/ericgrimson/Dropbox/MIT/Lecture2010New/Lecture9')

In [24]: c = Coordinate(3,4)
In [25]: print(c)
<3, 4>
In [26]: print(type(c))
<class '__main__.Coordinate'>
In [27]: print(isinstance(c, Coordinate))
True
In [28]: runfile('/Users/ericgrimson/Dropbox/MIT/Lecture2010New/Lecture9/coordinate.py',
              wdir='/Users/ericgrimson/Dropbox/MIT/Lecture2010New/Lecture9')
In [29]: c = Coordinate(3,4)
In [30]: origin = Coordinate(0,0)
In [31]: c - origin
Out[31]: <__main__.Coordinate at 0x11935f9e0>
In [32]: foo = c - origin
In [33]: print(foo)
<3, 4>
In [34]: 
```

the __ operator is overridden
(careful)

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

```

In [36]: twoThirds = fraction(2,3)
In [37]: print(oneHalf)
1 / 2
In [38]: print(twoThirds)
2 / 3
In [39]: oneHalf.getNumer()
Out[39]: 1
In [40]: new = oneHalf + twoThirds
In [41]: print(new)
7 / 6
In [42]: threeQuarters = fraction(3,4)
In [43]: print(threeQuarters)
3 / 4
In [44]: secondNew = twoThirds - threeQuarters
Traceback (most recent call last):
File "<ipython-input-44-d0d7943dac23>", line 1, in <module>
secondNew = twoThirds - threeQuarters
NameError: name 'two' is not defined
In [45]: secondNew = twoThirds - threeQuarters
In [46]: print(secondNew)
-1 / 12
In [47]: 
```

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

The left window shows the source code for an `intSet` class. The right window shows the console output of running the code.

```

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)/Lecture2816New/Lecture9/integer_set.py',
wdir='/Users/ericgrimson/Dropbox (MIT)/Lecture2816New/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 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>)`
- \downarrow for human beings to read \downarrow for programmers & interpreters

```

>>> # __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

```

remember this

```

>>> 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)
...
...
>>> my_car = Car('red', 37281)
>>> my_car
Car(red, 37281)
>>> str(my_car)
'Car(red, 37281)'
>>> print(my_car)
Car(red, 37281)
>>> 

```

* * rebult 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. \downarrow \downarrow
 \swarrow clients \searrow programmers \Rightarrow `str(<obj>)/print(<obj>)` \Rightarrow `<objectName>`

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.

⑯ 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:

goods = 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):

```
f(*a)
args: ('a',)
f(**a)
args: {}
args: ('a')
args: ('a', 'a')
args: ('a', 'a')
f(1,2,3)
args: (1, 2)
args: (1, 2, 3)
```

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

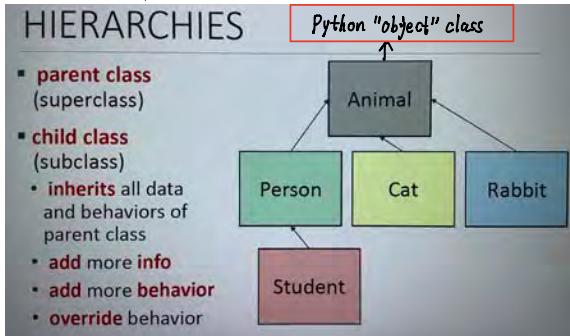
⑯ Implementing the class vs. Using the class (separate development)

⑰ in Python, better to set a variable's default value "=None" if we do NOT use it temporarily.

⑱ Python NOT good at information hiding – use Getters & Setters !

Part B - Hierarchy & Inheritance

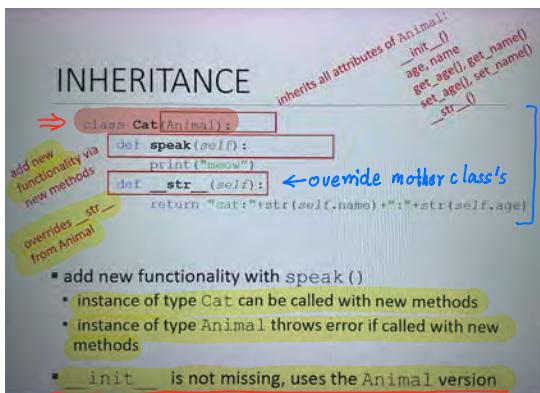
① an example to explain basics



INHERITANCE

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class object
- implements basic operations in Python, like binding variables, etc
- new object class
- inherits properties of underlying Python object class



USING THE HIERARCHY

```
In [31]: jelly = Cat(1)
In [32]: jelly.set_name('JellyBelly')
In [33]: print(jelly)
cat:JellyBelly:1

In [34]: print(Animal.__str__(jelly))
Animal:JellyBelly:1

In [35]: blob = Animal(1)
In [36]: print(blob)
animal:None:1

In [37]: blob.set_name()
In [38]: print(blob)
animal:None:1
```

inherits method from Animal
cat `str` method shadows
could explicitly recover underlying Animal method
can change values of attributes of an instance

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 own initializer, which can be completely new or inherit part of parent's

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, name)
        self.name = name
        self.age = age
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("Hello")
    def age_diff(self, other):
        # alternate way: diff = self.age - other.age
        diff = self.get_age() - other.get_age()
        if self.age > other.age:
            print(f"{self.name} is {diff} years older than {other.name}")
        else:
            print(f"{self.name} is {diff} years younger than {other.name}")
    def __str__(self):
        return f"person:{self.name}:{self.age}"
```

USING THE HIERARCHY

In [42]: eric = Person('Eric', 45)
 In [43]: john = Person('John', 55)
 In [44]: eric.speak()
 Hello

In [45]: eric.age_diff(john)
 Eric is 10 years younger than John

In [46]: Person.age_diff(john, eric)
 John is 10 years older than Eric

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("I have homework")
        elif 0.25 <= r < 0.5:
            print("I need sleep")
        elif 0.5 <= r < 0.75:
            print("I should eat")
        else:
            print("I am watching tv")
    def __str__(self):
        return f"student:{self.name}:{self.age}:{self.major}"
```

initializer inheriting same style

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

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

⑦ whenever we code a subclass, consider "shall we change higher classes' attribute(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):
        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.b = 2 ← Q5
    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):
    def __init__(self):
        C.__init__(self)
        B.__init__(self)
        self.d = 6
    def z(self):
        print("D.z")
```

obj = D()

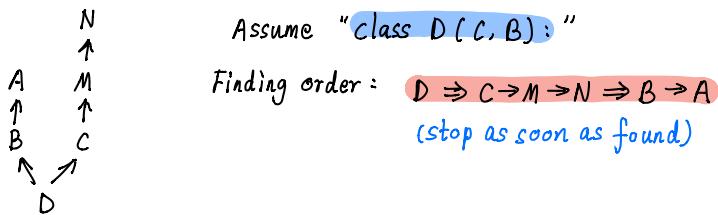
then what is printed by each of the following statements?

1. print(obj.a)
2. print(obj.b)
3. print(obj.c)
4. print(obj.d)
5. obj.x()
6. obj.y()
7. obj.z()

Answers:

- 1. 2
- 2. 3
- 3. 5
- 4. 6
- 5. A.x
- 6. C.y
- 7. D.z

- ⑩ Another illustration of ⑨ finding methods:



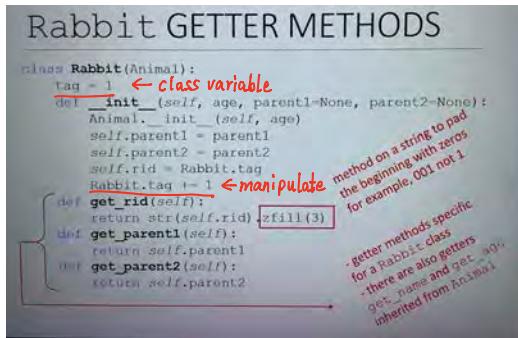
- ⑪ "up to down codes' sequence" is extremely important - if in the same file,
super classes MUST be in front of inheriting subclasses.

- 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 create < "inside __init__(self,...)" method: <className>. <varName> = xx
outside all methods (usually in the beginning): <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 of = 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
```

```
In [1]: foo = genTest()
In [2]: foo.__next__()
"foo" is a generator object
>>> foo.__next__()
2

>>> foo.__next__()
results in a StopIteration exception
```

Execution will proceed in body of foo, until reaches first yield statement; then returns value associated with that statement

Execution will resume in body of foo at point where stopped; until reaches next yield statement; then returns value associated with that statement

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 subsequent 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 ④'s inefficiency by Generator

FIX TO GRADES CLASS

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

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

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 i th letter in the alphabet.

② Open, Read and Deal with .txt files

```

testpy  * pypy  * story.txt  * words.txt
1> pypy
1 import string
2
3 ### DO NOT MODIFY THIS FUNCTION ##
4 def load_words(file_name):
5     ...
6     file_name (string): the name of the file containing
7     the list of words to load
8
9     Returns: a list of valid words. Words are strings of lowercase letters.
10
11    Depending on the size of the word list, this function may
12    take a while to finish.
13
14    print('Loading word list from file...')
15    # inFile: file
16    in_file = open(file_name, 'r')
17    # line: string
18    line = in_file.readline() ONLY useful for small data
19    # word_list: list of strings
20    word_list = line.split() ("for line in in_file" is better)
21    print(' ', len(word_list), 'words loaded.')
22    in_file.close()
23    return word_list
24

```

```

testpy  * pypy  * story.txt  * words.txt
1> pypy
24
25 ### DO NOT MODIFY THIS FUNCTION ##
26 def is_word(word_list, word):
27     ...
28     Determines if word is a valid word, ignoring
29     capitalization and punctuation
30
31     word_list (list): list of words in the dictionary.
32     word (string): a possible word.
33
34     Returns: True if word is in word_list, False otherwise
35
36     Example:
37     >>> is_word(word_list, 'bat') returns
38     True
39     >>> is_word(word_list, 'asdf') returns
40     False
41
42     word = word.lower()
43     word = word.strip(" !@#$%^&*()-_=+{}[]|\';':<>?,.-/")
44     return word in word_list
45
46 ### DO NOT MODIFY THIS FUNCTION ##
47 def get_story_string():
48     ...
49     Returns: a joke in encrypted text.
50
51     f = open("story.txt", "r")
52     story = str(f.read())
53     f.close() for small data
54     return story

```

③ for "Getter" methods, 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__(self)`" inheritance, better use `self.<methodName>(xx,...)` to call all methods within the hierarchy.

In fact, we can use >`super().__init__(self, 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 $\begin{cases} \text{with a timer?} \\ \text{count the operations?} \end{cases}$ 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

COMPLEXITY CLASSES

- $O(1)$ denotes constant running time
- $O(\log n)$ denotes logarithmic running time
- $O(n)$ denotes linear running time
- $O(n \log n)$ denotes log-linear running time
- $O(n^c)$ denotes polynomial running time (c is a constant)
- $O(c^n)$ denotes exponential running time (c is a constant being raised to a power based on size of input)

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

LOGARITHMIC COMPLEXITY

```
def intToStr(i):  
    digits = '0123456789'  
    if i == 0:  
        return '0'  
    res = ''  
    while i > 0:  
        res = digits[i%10] + res  
        i = i//10  
    return result
```

- only have to look at loop as no function calls
- within while loop, constant number of steps
- how many times through loop?
 - how many times can one divide i by 10?
 - $O(\log(i))$

this is independent of input size.

TRICKY COMPLEXITY

```
def h(n):  
    """ assume n an int >= 0 """  
    answer = 0  
    s = str(n)  
    for c in s:  
        answer += int(c)
```

$O(\log n)$
linear $O(|len(s)|)$
but what in terms of input n ?

- adds digits of a number together
- tricky part
 - convert integer to string
 - iterate over length of string, not magnitude of input n
 - think of it like dividing n by 10 each iteration
- $O(\log n)$ – base doesn't matter

- 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
```

- Best case: $O(1)$
- Worst case: $O(1) + O(n) + O(1) \rightarrow O(n)$

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

- 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 e not in res:
            res.append(e)
    return res
```

first nested loop takes $\text{len}(L1)*\text{len}(L2)$ steps
second loop takes at most $\text{len}(L1)$ steps
latter term overwhelmed by former term
 $O(\text{len}(L1)*\text{len}(L2))$

Remark: immutable data concatenation repeatedly 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 = [] ← return all subsets of L
    if len(L) == 0:
        return [()]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

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 $2^{n-1} + 2^{n-2} + \dots + 2^0$ 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!

A recursion tree diagram for the recursive Fibonacci function. The root node is labeled n . It branches into two children, each labeled $n-1$, which further branch into $n-2$ and $n-2$ respectively, and so on. Red arrows point from the text "2 recursive calls at same time" to the branching nodes, and from "this is bad!" to the tree structure, illustrating the inefficiency of the exponential time complexity.

(13) when the input is a List

WHEN THE INPUT IS A LIST...

```
def sum_list(L):
    total = 0
    for e in L:
        total = total + e
    return total
```

- $O(n)$ where n is the length of the list
- $O(\text{len}(L))$
- must **define what size of input means**
 - previously it was the magnitude of a number
 - here, it is the length of list

(14) Complexity of common Python Function

Lists: $n = \text{len}(L)$	Dictionaries: $n = \text{len}(d)$
• index $O(1)$	• worst case ✓
• store $O(1)$	• index $O(n)$
• length $O(1)$	• store $O(n)$
• append $O(1)$	• length $O(n)$
• == $O(n)$	• delete $O(n)$
• remove $O(n)$	• iteration $O(n)$
• copy $O(n)$	• average case
• reverse $O(n)$	• index $O(1)$
• iteration $O(n)$	• store $O(1)$
• in list $O(n)$	• delete $O(1)$
	• iteration $O(n)$

(15) Normally, the minimum operations (\approx 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.

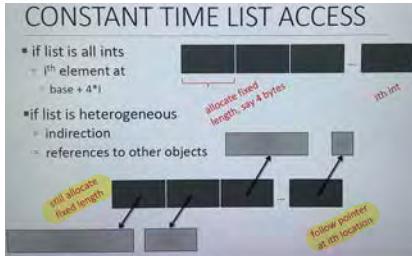
(2) Linear Search : brute force search ; Lists does NOT have to be sorted

LINEAR SEARCH ON UNSORTED LIST

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

- must look through all elements to decide it's not there
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
- overall complexity is $O(n)$ – where n is $\text{len}(L)$

(speed up a little by returning True here, but speed up doesn't impact worst case)



LINEAR SEARCH ON SORTED LIST

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- must only look until reach a number greater than e
- $O(\text{len}(L))$ for the loop * $O(1)$ to test if $e == L[i]$
- overall complexity is $O(n)$ – where n is $\text{len}(L)$

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 COMPLEXITY ANALYSIS

- complexity is $O(\log n)$ – where n is $\text{len}(L)$

BISECTION SEARCH IMPLEMENTATION 1

```
def bisect_search1(L, e):
    if L == []:
        return False
    elif len(L) == 1:
        return L[0] == e
    else:
        half = len(L)//2
        if L[half] > e:
            return bisect_search1(L[:half], e)
        else:
            return bisect_search1(L[half:], e)
```

copy is bad signal

Annotations explain the complexity of each part of the code:

- first if statement: constant $O(1)$
- second if statement: constant $O(1)$
- assignment: constant $O(1)$
- comparison: constant $O(1)$
- slicing: NOT constant, copies list
- recursive call: NOT constant
- recursive call: NOT constant
- recursive call: NOT constant

BISECTION SEARCH IMPLEMENTATION 2 $O(\log n)$

```

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
indices
NOT constant

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
- $\rightarrow O(n \log n)$
- $\rightarrow O(n)$ 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~~
- $\rightarrow O(\log n)$

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.

$O(n^2)$

COMPLEXITY OF BUBBLE SORT

```

def bubble_sort(L):
    swap = False
    while not swap: O(len(L))
        swap = True
        for i in range(1, len(L)): O(len(L))
            if L[i-1] > L[i]:
                swap = False
                temp = L[i]
                L[i] = L[i-1]
                L[i-1] = temp
    # inner for loop is for doing comparisons
    # outer while loop is for doing multiple passes until no more swaps
    # O(n^2) 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

$O(n^2)$

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

```

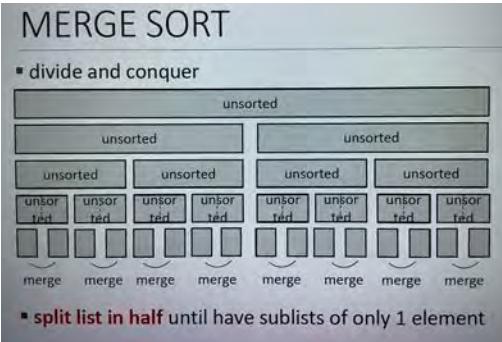
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]: len(L) times
                L[suffixSt], L[i] = L[i], L[suffixSt] len(L) - suffixSt times
        suffixSt += 1
2 layers loops
phthonic swap

```

- outer loop executes $\text{len}(L)$ times
- inner loop executes $\text{len}(L) - i$ times
- complexity of selection sort is $O(n^2)$ where n is $\text{len}(L)$

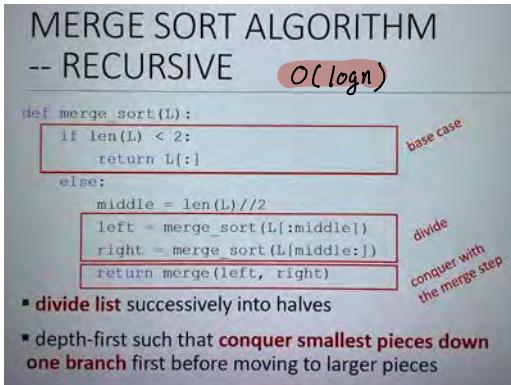
④ Merge Sort : divide-and-conquer approach → split into 2 parts and then merge (recursively)

$O(n \log n)$



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]



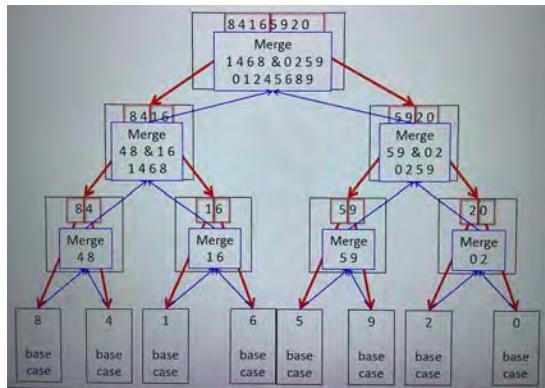
MERGING SUBLISTS STEP $O(n)$

```
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 $\text{len}(L)$
- at second recursion level
 - $n/4$ elements in each list
 - two merges → $O(n)$ where n is $\text{len}(L)$
- each recursion level is $O(n)$ where n is $\text{len}(L)$
- dividing list in half with each recursive call
 - $O(\log(n))$ where n is $\text{len}(L)$
- overall complexity is $O(n \log(n))$ where n is $\text{len}(L)$

Unit 7 - Plotting

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

① 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(<listname>, <listname>)"
 x-axis y-axis "pylab.show()" show a separate window of plotting result

A screenshot of a Jupyter Notebook interface. On the left, a code editor shows Python code for generating four types of plots: linear, quadratic, cubic, and exponential. The code uses lists to store sample data and then plots them against an x-axis. An annotation 'x-axis v-axis' points to the x-axis label in the code. On the right, an 'Interactive' window displays four separate plots: a linear function (a straight line), a quadratic function (a parabola), a cubic function (an S-shape), and an exponential function (a curve that increases rapidly). A red annotation 'this is the Jupyter interactive window' points to this window.

```
#106
1 import pylab as plt
2
3 mySamples= []
4 myLinear= []
5 myQuadratic= []
6 myCubic= []
7 myExponential= []
8
9 for i in range(0, 30):
10     mySamples.append(i)
11     myLinear.append(i)
12     myQuadratic.append(i**2)
13     myCubic.append(i**3)
14     myExponential.append(1.5**i)
15
16 plt.plot(mySamples, myLinear) x-axis v-axis
17 plt.plot(mySamples, myQuadratic)
18 plt.plot(mySamples, myCubic)
19 plt.plot(mySamples, myExponential)
20 plt.show()
21
22 #%%
```

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

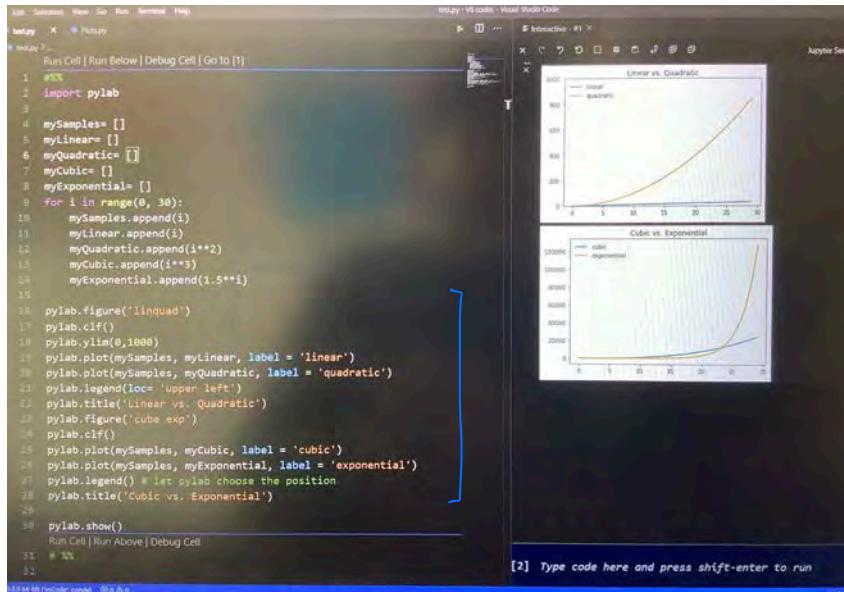
clean up previous created windows (avoid figure formatting error) "pylab.clf()"

A screenshot of a Jupyter Notebook interface. On the left, a code editor shows Python code for creating two plots: a linear function and an exponential function. The code uses the 'figure' command to create separate windows for each plot. Annotations indicate the first 'figure' call as 'create a figure first' and the subsequent 'plot' call as 'plot after setting'. On the right, two separate windows are shown: one for the linear function (a straight line) and one for the exponential function (a curve that increases rapidly). An annotation 'usually after ".figure(xx)"' points to the 'clf()' call in the code, which is used to clear the previous figure.

```
1 import pylab
2
3 mySamples= []
4 myLinear= []
5 myQuadratic= []
6 myCubic= []
7 myExponential= []
8
9 for i in range(0, 30):
10     mySamples.append(i)
11     myLinear.append(i)
12     myQuadratic.append(i**2)
13     myCubic.append(i**3)
14     myExponential.append(1.5**i)
15
16 pylab.figure('lin') ← create a figure first
17 pylab.clf()
18 pylab.title('Linear')
19 pylab.xlabel('sample points')
20 pylab.ylabel('linear function')
21 pylab.plot(mySamples, myLinear) ← plot after setting
22
23 pylab.figure('expo')
24 pylab.clf()
25 pylab.ylabel('quadratic function')
26 pylab.plot(mySamples, myExponential)
27
28 pylab.show()
```

⑤ Comparing Plots :

- 5.1 changing Limits on axes `pylab.xlim (<minNum>, <maxNum>) ; pylab.ylim (<minNum>, <maxNum>)`
- 5.2 Overlaying plots write ".plot (a,b) & .plot (c,d) & ..." under same ".figure (<>)"
- 5.3 Adding more parameters `pylab.plot (<list1>, <list2>, label = <strName>)`
- 5.4 set a location for 5.3 label(s) " `pylab.legend (loc = <location Argument>)`"
 ↪ 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
 '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

3rd argument (optional)
 ↴

CHANGING DATA DISPLAY

```

plt.figure('lin quad')
plt.clf()
plt.plot(mySamples, myLinear, 'b-' , label = 'linear')
plt.plot(mySamples, myQuadratic, 'r o', label = 'quadratic')
plt.legend(loc = 'upper left')
plt.title('Linear vs. Quadratic')

plt.figure('cube exp')
plt.clf()
plt.plot(mySamples, myCubic, 'g^', label = 'cubic')
plt.plot(mySamples, myExponential, 'r--',label = 'exponential')
plt.legend()
plt.title('Cubic vs. Exponential')

```

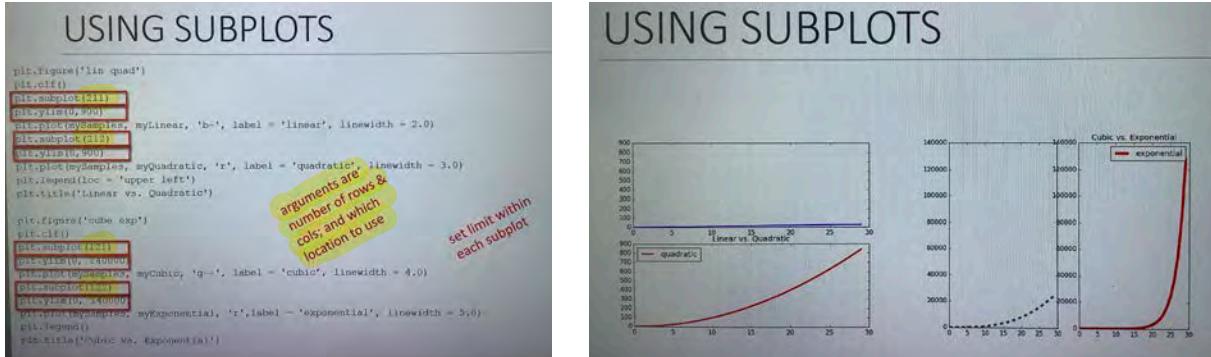
string specifies color and style
see documentation for choices of color and style



- 6.2 Line width (5th parameter, 线的粗细) in

`".plot (<list1>, <list2>, 'ox', label=<>, linewidth = <float>)"`

6.3 Using Subplots "pylab.subplot('0xT')" '0': # of rows; 'x': # of cols; 'T': location

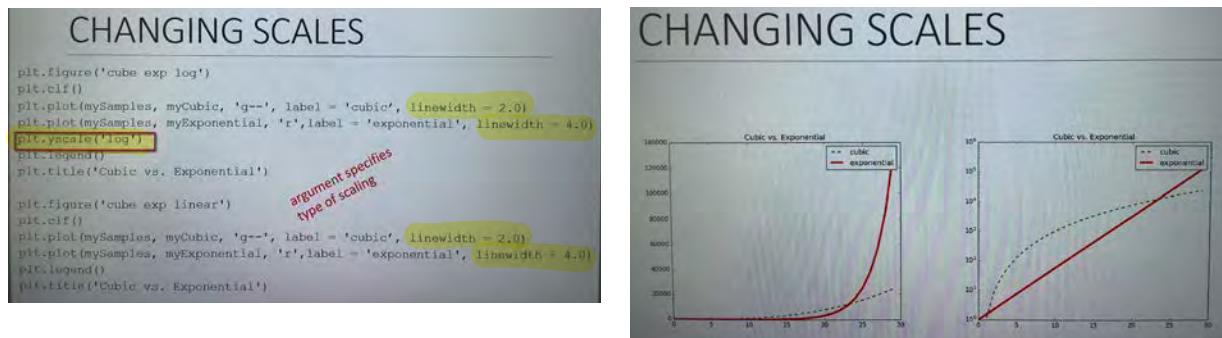


6.4 Changing Scales "pylab.xscale(<argument>)" "pylab.yscale(<argument>)"

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)

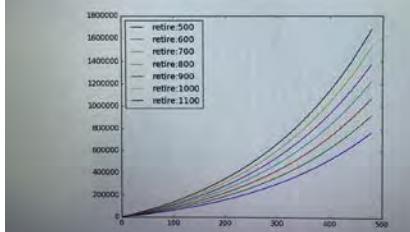
AN EXAMPLE: compound interest

monthly saving

```
def retire(monthly, rate, terms):
    savings = [0]
    base = [0]
    mRate = rate/12
    for i in range(terms):
        base += [1]
        savings += [savings[-1]*(1 + mRate) + monthly]
    return base, savings
```

↑
yearly rate
of months
x-axis, y-axis

DISPLAYING RESULTS vs. MONTH



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

DISPLAYING RESULTS vs. MONTH savings

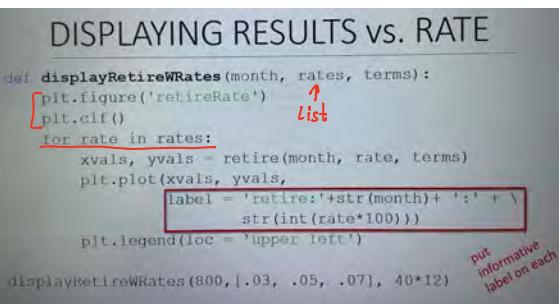
List single value

```
def displayRetireWMonthlies(monthlies, rate, terms):
    plt.figure('retireMonth')
    plt.clf()
    for monthly in monthlies:
        xvals, yvals = retire(monthly, rate, terms)
        plt.plot(xvals, yvals, label = 'retire:' + str(monthly))
    plt.legend(loc = 'upper left')

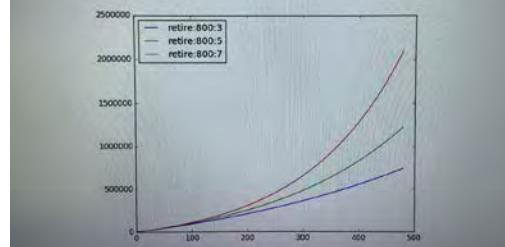
displayRetireWMonthlies([500, 600, 700, 800, 900,
    1000, 1100], .05, 40* 12)
```

clear frame so can reuse
put informative label on each

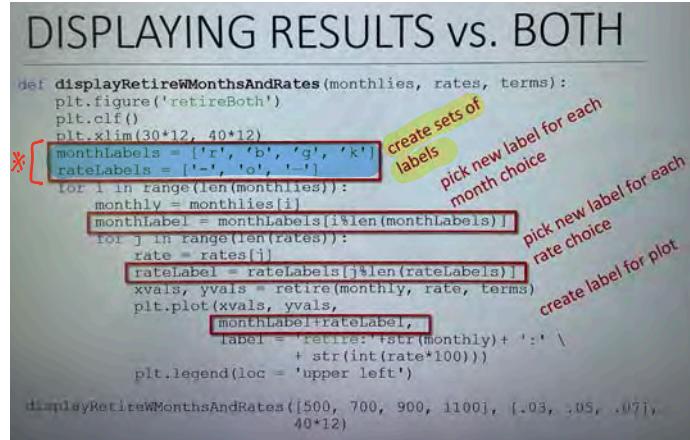
② vs. yearly rates (input: list - rates)



DISPLAYING RESULTS vs. RATE



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



DISPLAYING RESULTS vs. BOTH

