Q)  Is Python a strongly or weakly typed language?

A) Python is strongly, dynamically typed.

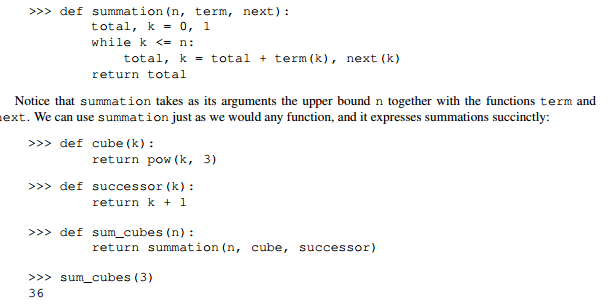
Python is strongly typed, because **objects don't change type**. Python is dynamic typed, because **we pass around references and don't check the type until the last possible minute**.

Q)  Difference between using the def keyword and lambda expressions

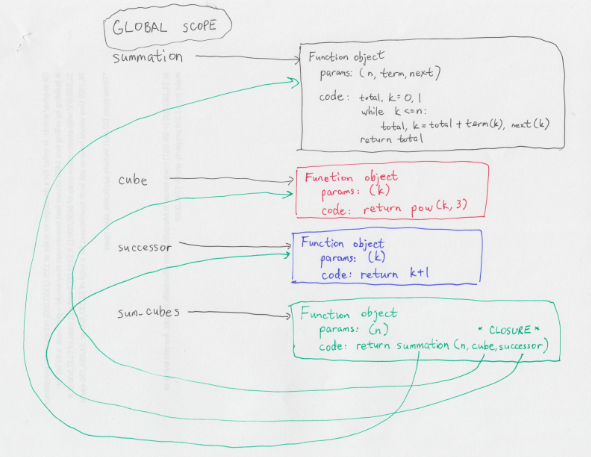
A) def is a statement, while lambda is an expression. Evaluating a def statement will have a side effect; namely, it creates a new function binding in the current environment. On the other hand, evaluating a lambda expression will not change the environment unless we do something with this expression. For instance, we could assign it to a variable or pass it in as a function argument.

----------------------------------------------------------------------------------------------------------------------------------

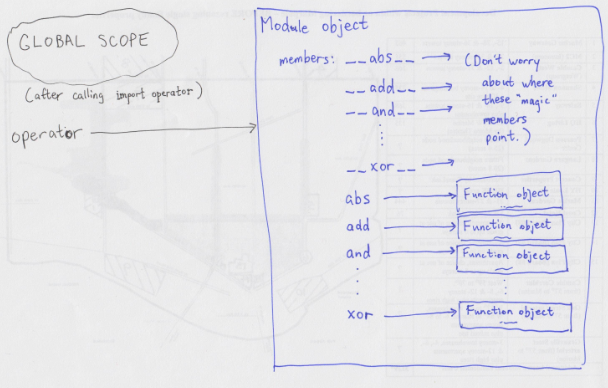
For the code,



Higher order functions look like this:



Importing modules look like this:



-----------------------------------------------------------------------------------------------------------------------------------

Q) How do I visualize local frame and global frame in python Environment?

A) *global frame* belongs to \_\_main\_\_ module(kind of) where \_\_main\_\_ module is the name of the module that interpreter loads for your program with default members.

>>> globals()

{'\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None}

>>>

So, when i say kind of, I mean, global frame actually belongs to interpreter space and the global frame that is visible to you has the things that are available to your program.

So when we enter in interactive mode or say ‘python file.py’ that means, we are in \_\_main\_\_ module already.

Being in \_\_main\_\_ module mean whatever you type is evaluated in the scope of this module, and whatever you assign populates this module, that's what can be described as being in that module.

Another important point is \_\_main\_\_ is the name of the module object we are in. We don’t have access to reference variable that points to \_\_main\_\_ module object.

*Local frame is:*

Say if am calling square.\_\_call\_\_(3) then am in Activation Record of \_\_call\_\_() of square object which is of class ’function’

>>>operator.\_\_dict\_\_ gives the dictionary view of operator module object.

Q) Query on handling TAB character

A) Note that TAB is same as other characters. \r\n may appear in notepad as a 5 pixel gap between lines but in hyperterminal it's probably a 9 pixel gap. Similarly, \n may appear in vim as a 6 pixel gap between lines but in notepad it's a square box (or is that ^M? I don't remember). Don't confuse what the UI displays on screen with the data on disk.

Python error:

TabError: inconsistent use of tabs and spaces in indentation

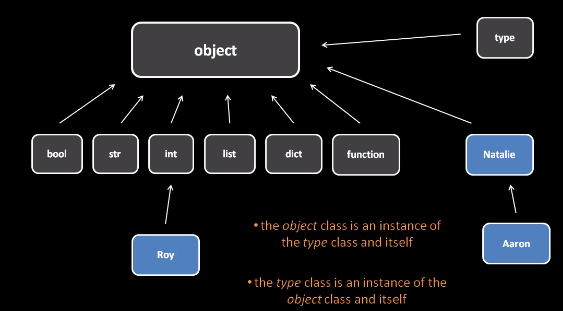
Python interprets a tab as eight spaces. So if your text editor displays them differently you need double check. For this reason, python programmers discourage mixing tabs and spaces in code. I believe the recommendation is to always use spaces? So in an editor like notepad you need to use the spacebar. In vi or emacs or visual studio you can configure the preferences so that your editor inserts spaces (not tab) when you press the tab key.

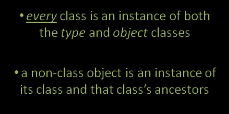
What I mean is that in most editors 4 space characters and one tab characters look the same but they're not the same on disk. In most programming languages it doesn't matter but in python it does.

----------------------------------------------------------------------------------------------------------------------------------------

Q) Query on 'object' class & 'type' class in python?

A)





Your understanding is basically right. type and object are special in that they are the base of the type hierarchy. Everything is an instance of object. Every type/class is an instance of type. So type is an instance of object and object is also an instance of type.

This means that the inheritance relationship cannot really be represented as a tree, because there is a cycle: type and object are instances of each other. This kind of mutual inheritance is not normally possible, but that's the way it is for these fundamental types in Python: they break the rules. The diagram is somewhat misleading since it shows type as inheriting from object but not vice versa. Really, the arrow between type and object should be double-headed, indicating the inheritance goes both ways.

It's important to distinguish between what types/classes *inherit* from and what they are *instances* of, although this isn't directly represented in that diagram. A class or type (like int or Natalie) is a *subclass* of object, but it is an *instance* of type. The two statements that you refer to in Question 3 relate to this. The object type is an instance of object, because everything is an object; object is also an instance of type, because object is a type (aka a class). Likewise, type is an instance of object, because everything is an object; and type is also an instance of type, because type is a type (it is the type of types, and the type of user-defined classes).

There is also an inaccuracy in the diagram: bool is not actually a direct subclass of object, rather it is a subclass of int (which is a direct subclass of object).

issubclass(type, object)

isinstance(type,object)

Something can only be a subclass of object, if it’s a type itself. And as such it would already be an instance of type which is a subclass of object.

isinstance(object, type)

Because object is a type; everything is an instance of object, so object needs to be a type.

isinstance(type, type)

Because type is the type of all types. And as type is a type, it(object) is also an instance of type.

>>> isinstance(object, object)

 as object is part of the “everything”, it is an instance of object.

Q) How can I do a line break (line continuation) in Python while coding?

A)

What is the line? You can just have arguments on the next line without any problems:

a = dostuff(blahblah1, blahblah2, blahblah3, blahblah4, blahblah5,

blahblah6, blahblah7)

Otherwise you can do something like this:

if a == True and \

b == False

Check the [style guide](http://www.python.org/dev/peps/pep-0008/) for more information.

From your example line:

a = '1' + '2' + '3' + \

'4' + '5'

Or:

a = ('1' + '2' + '3' +

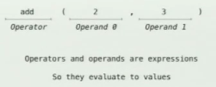
'4' + '5')

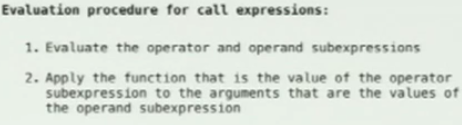
Note that the style guide says that using the implicit continuation with parentheses is preferred, but in this particular case just adding parentheses around your expression is probably the wrong way to go.

Building Abstractions, *CS61A*

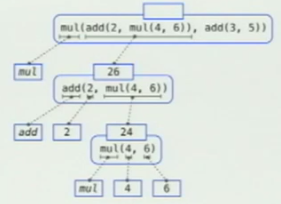
+++++++++

An **expression** describes a computation and evaluates to a value.



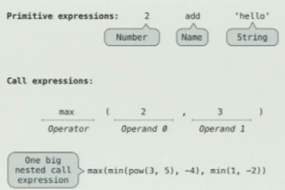


Evaluating Nested Expressions:

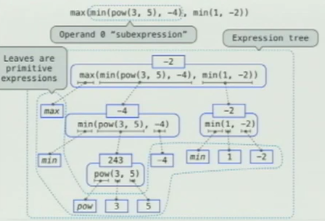


The above rule of evaluating expressions is common among all programming languages, But there could be some unknown exception to this rule.

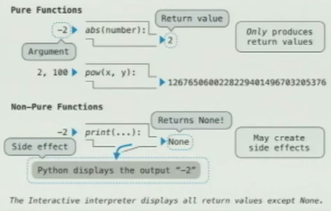
**Types of Expressions:**

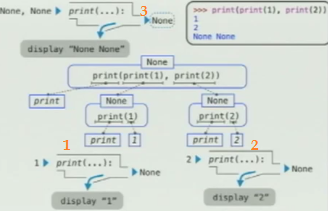


**Expression Tree:**



**print** function in python:





It doesn’t mean that all non-pure functions return **None**. Non-pure function could return anything. Function is non-pure, because it has this backdoor, that allows it to have side-effect.

**++++++++**

**Naming(Name binding) in python**







Assignment statements,

import statements

&

def statements change the environment.

Named values are simple means of **abstraction**.

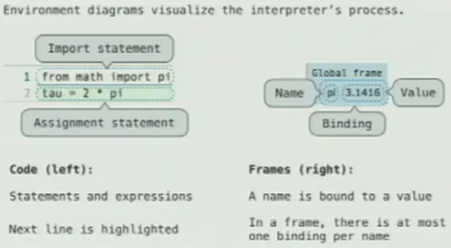
Named expressions are a more powerful means of **abstraction**, which are nothing but functions.

**Order Of Evaluation in assignment statement**:

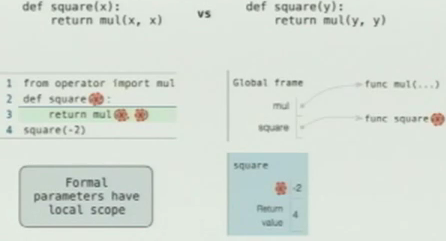


We evaluate the first one, then we evaluate the second one, before ever changing any Name binding and then we bind all the names for these values.

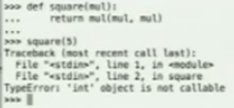
**Environment Diagrams:**



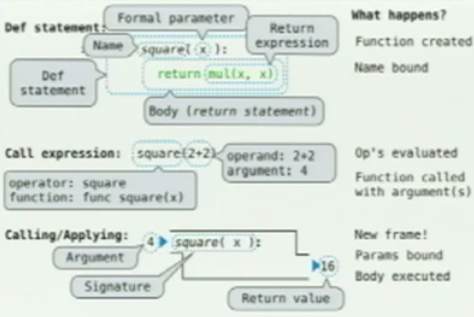
**Formal parameters:**



**One example:**



**Life Cycle of a User-Defined Function**



Function only returns one value. It can be reference to a tuple that contains references to two other objects.

--------------------------------------------------------------------------------------------------------------------------------

Q) Python’s **return**, **return None**, and no return at all

A) Using return None.

This tells that the function is indeed meant to return a value for later use, and in this case it returns None. This value None can then be used elsewhere. return None is never used if there are no other possible return values from the function.

In the following example, we return person's mother if the person given is a human. If it's not a human, we return None since the "person" doesn't have a mother (let's suppose it's not an animal or so).

def get\_mother(person):

if is\_human(person):

return person.mother

else:

return None

Using return.

This is used for the same reason as break in loops. The return value doesn't matter and you only want to exit the whole function. It's extremely useful in some places, even tho you don't need it that often.

We got 15 prisoners and we know one of them has a knife. We loop through each prisoner one by one to check if they have a knife. If we hit the person with a knife, we can just exit the function cause we know there's only one knife and no reason the check rest of the prisoners. If we don't find the prisoner with a knife, we raise an the alert. This could be done in many different ways and using return is probably not even the best way, but it's just an example to show how to use return for exiting a function.

def find\_prisoner\_with\_knife(prisoners):

for prisoner in prisoners:

if "knife" in prisoner.items:

prisoner.move\_to\_inquisition()

return # no need to check rest of the prisoners nor raise an alert

raise\_alert()

Note: You should never do var = find\_prisoner\_with\_knife(), since the return value is not meant to be caught.

Using no return at all.

This will also return None, but that value is not meant to be used or caught. It simply means that the function ended successfully. It's basically the same as return in void functions in languages such as C++ or Java.

In the following example, we set person's mother's name, and then the function exits after completing successfully.

def set\_mother(person, mother):

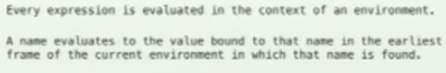
if is\_human(person):

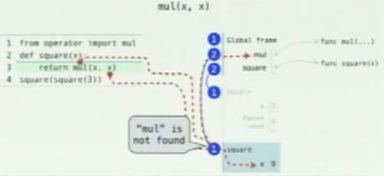
person.mother = mother

Note: You should never do var = set\_mother(my\_person, my\_mother), since the return value is not meant to be caught.

------------------------------------------------------------------------------------------------------------------------------

**Names have no Meaning without Environments**

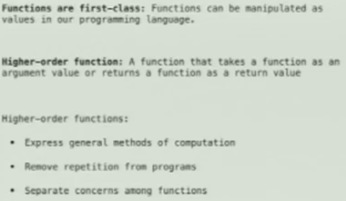




+++++++++

++++++++++++

**Higher order functions:**

****

Q) How function return multiple values in python?

A) def f(in\_str):

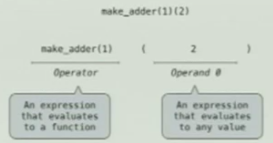
out\_str = in\_str.upper()

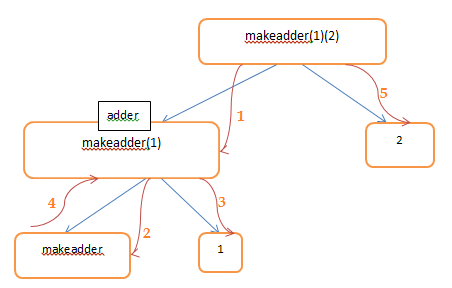
return True, out\_str # Creates tuple automatically

succeeded, b = f("a") # Automatic tuple unpacking

**Call Expressions as Operator Expressions:**

Call expression with a compound operator. Operand comes inside the parenthesis. Operator comes outside the parenthesis.





Lambda expressions and recursion:

One difference between using the def keyword and lambda expressions is that def is a statement, while lambda is anexpression. Evaluating a def statement will have a side effect; namely, it creates a new function binding in the current environment. On the other hand, evaluating a lambda expression will not change the environment unless we do something with this expression. For instance, we could assign it to a variable or pass it in as a function argument.

* If two implementations are equally clear, then shorter is better.
* When learning to write recursive functions, put the base cases first

Q) Mutable objects in python?

A) function type objects are mutable. Number/string types objects are immutable.

+++++++++++++++++

Q) name and object in python

A)

a = 42

def f():

def g():

return a

return f

def h(x):

def i():

return x

return i

old\_f = f

f = 666

result = old\_f()

+++++++++++++++++

Q) How multiple modules work together?

A) After i run >>> python -i hog.py, All definitions written in hog.py file are part of \_\_main\_\_ module.

In hog.py file, When i say,

from dice import four\_sided\_dice, six\_sided\_dice, make\_test\_dice

from ucb import main, trace, log\_current\_line, interact

 These functions are executing in the scope of ucb or dice module respectively but not in the scope of \_\_main\_\_ module. So, In above syntax, imports with names are like pointers to functions in the module outside \_\_main\_\_module. All this is verified using the value of global variable \_\_name\_\_.

++++++++++++++++++

Q) How \* works in below function definition?

def g(\*x):

for i in range(1, n):

x = f(\*x) if isinstance(x, tuple) else f(x)

return x

In def g(\*x), the \* operator lets g accept arbitrary arguments — x in this case will be a list containing any and all arguments. In f(\*x), the \* operator has the inverse effect of unpacking the argument list — instead of calling f with a single argument that is a list, let the first element of the list be the first argument, the second element of the list be the second argument, etc

+++++++++++++++++++

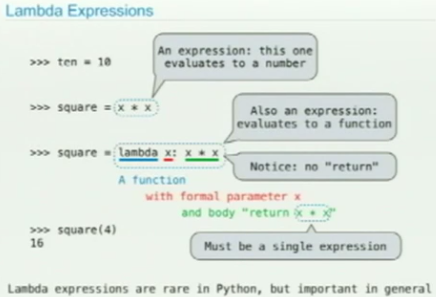
Q) How // operator works?

A)

|  |  |  |
| --- | --- | --- |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | 9//2 is equal to 4 and 9.0//2.0 is equal to 4.0 |

++++++++++++++++++++

**Lambda Expressions:**



Idea is, You don’t have to name intermediate values in a complicated expression like

>>> 0 + 1\*1 + 2\*2 + 3\*3

14

So, lambda x: x\*x expression creates a function object

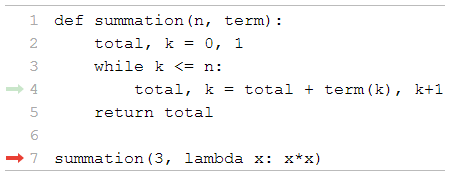
>>> summation(3, lambda x: x\*x)

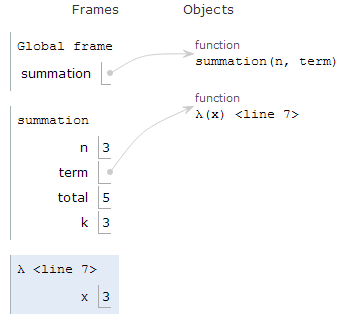
-> Primary advantage of lambda expressions is that we can create the functions without having to give it a name.

-> Only the def statement gives the function an intrinsic name.

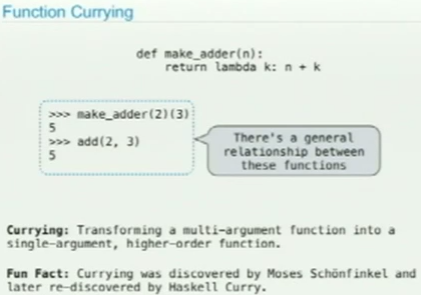
-> Javascript uses this lambda expression very often. Python rarely uses this feature.

-> If you have multiple expressions, better use def rather than using lambda because single expression looks complex.





**Function Currying:**



Why would we do this?

Sometimes we have a big program, where one part of the program knows only the first argument to the function you are going to call and create another function that could be passed on and on and later second argument is available. Hmm am still not clear.

Function decorator:

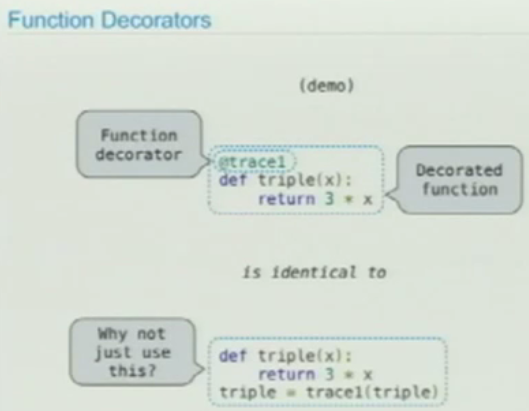
def trace1(f):

def traced(x):

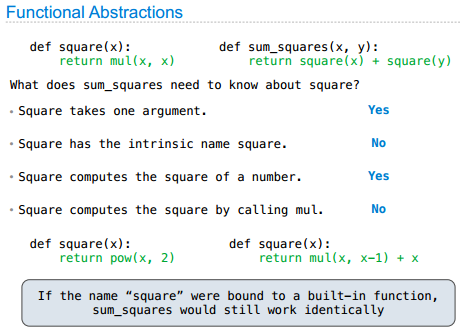
print('Called:', f, 'on', x)

return f(x)

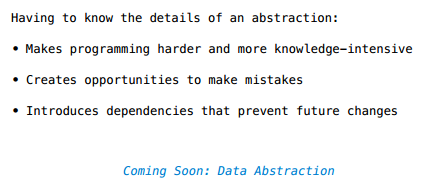
return traced



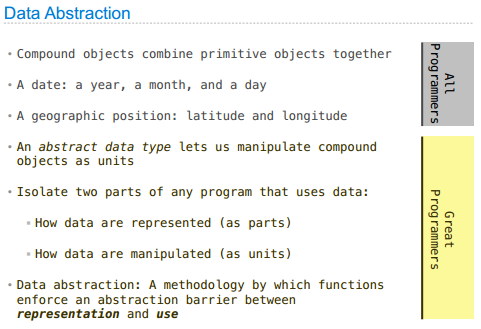
Functional abstraction:

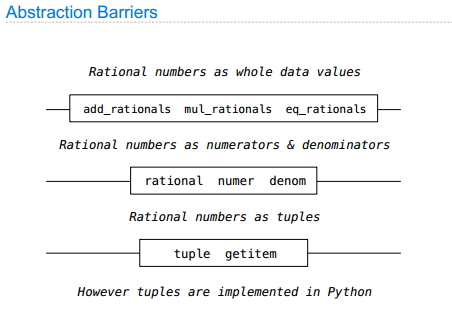


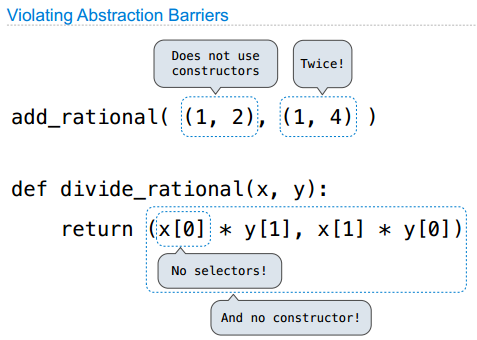
Data abstraction:

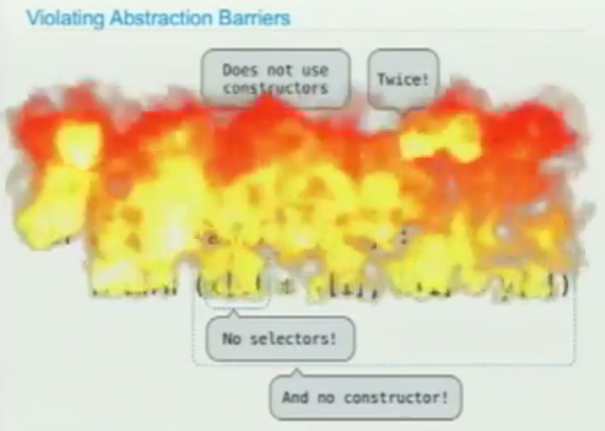


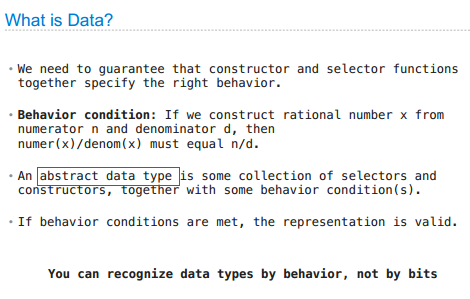
A methodology by which functions enforce an abstraction barrier between “how data values are used” and “how data values are represented”.







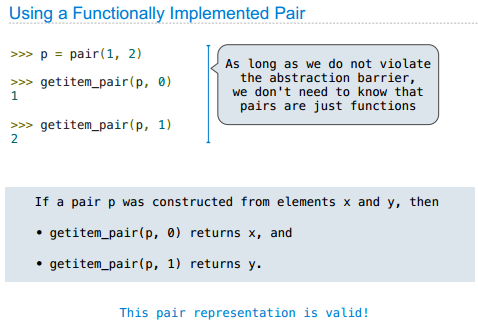


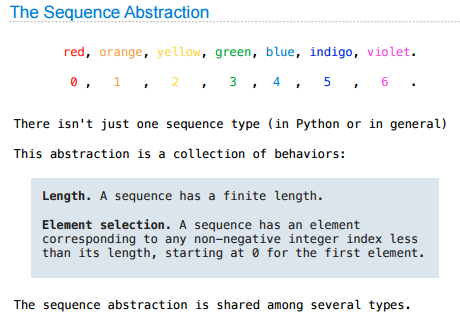


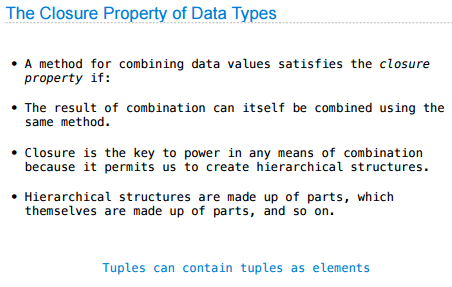
If I change the below definition, still the collection of constructor and selectors are ADT, Because it holds the behavior condition (Invariants).

def rational(n, d):

return(n \* 45, d \* 45)





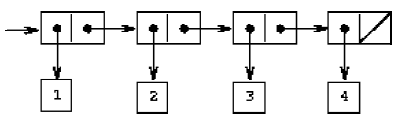


**Hierarchical data and the closure property:**

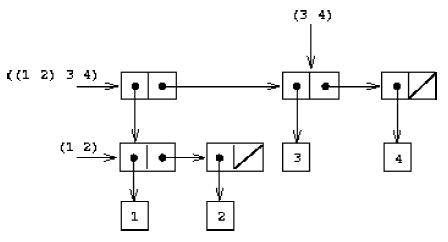
In general, an operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation. Closure is the key to power in any means of combination because it permits us to create *hierarchical* structures -- structures made up of parts, which themselves are made up of parts, and so on.

We take up the consequences of closure for compound data. We describe some conventional techniques for using pairs to represent sequences and trees.

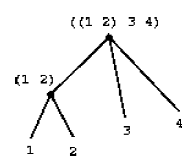
One of the useful structures we can build with pairs is a *sequence* -- an ordered collection of data objects.



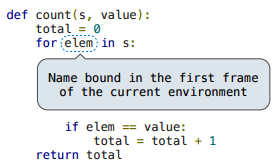
The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences. For example, we can regard the object ((1 2) 3 4) constructed by as a list of three items, the first of which is itself a list, (1 2).

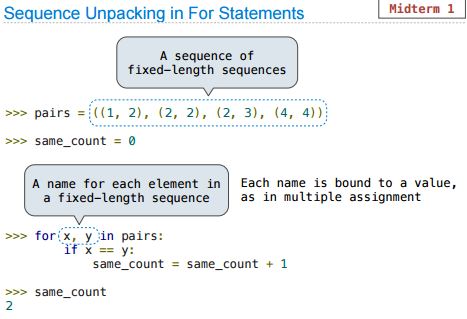


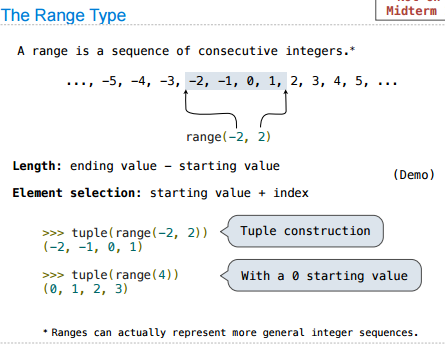
Another way to think of sequences whose elements are sequences is as *trees*. The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees.

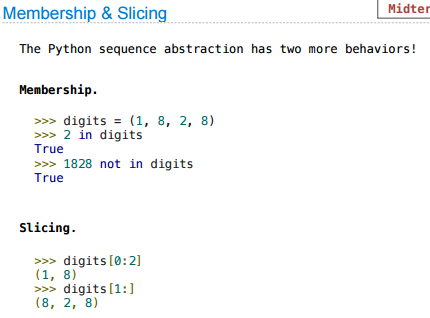


**Sequence iteration:**









>>> a=(1,2,3)

>>> from operator import not\_,contains

>>> contains(a,2)

True

>>> contains(a,4)

False

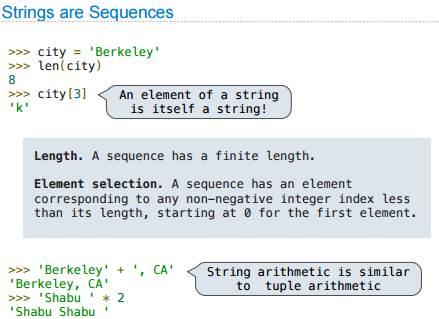
>>> not\_(contains(a,4))

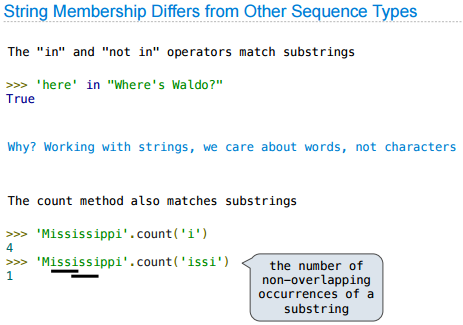
True

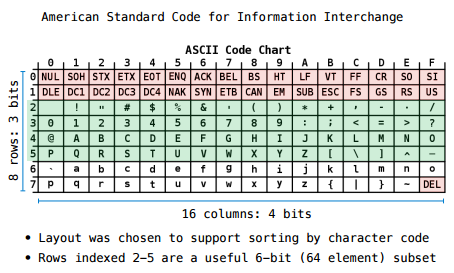
Advantage of Tuple over List

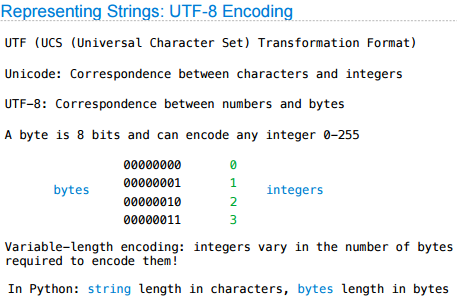
Tuples and list look quite similar except the fact that one is immutable and the other is mutable. We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes. There are some advantages of implementing a tuple than a list. Here are a few of them.

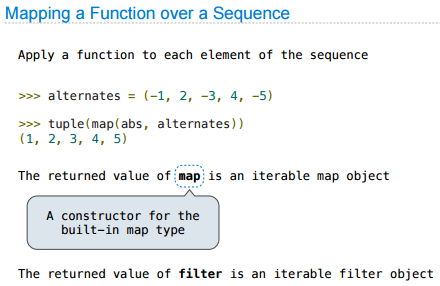
* Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
* Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
* If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

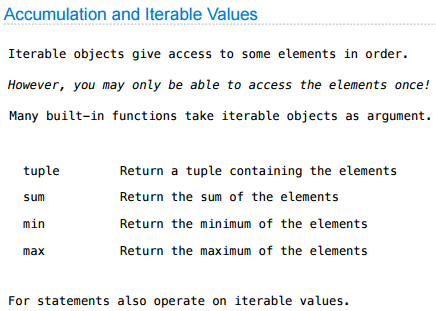


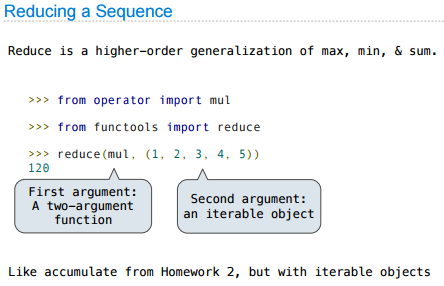


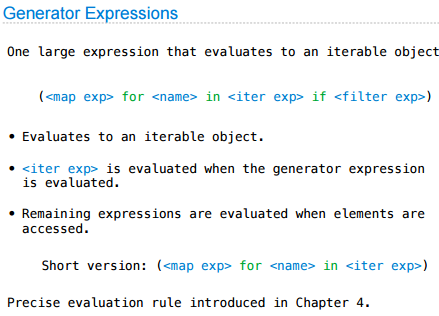


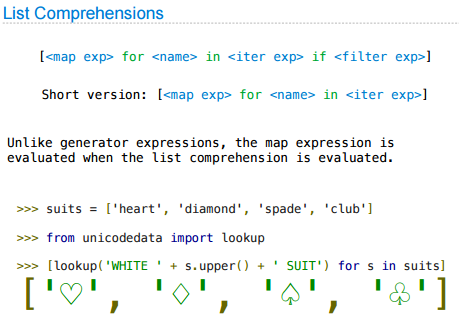


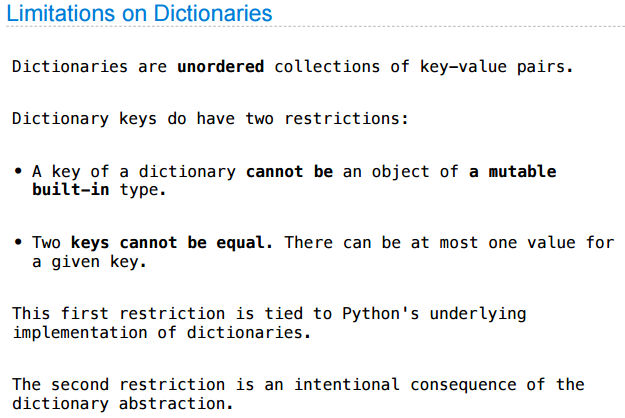


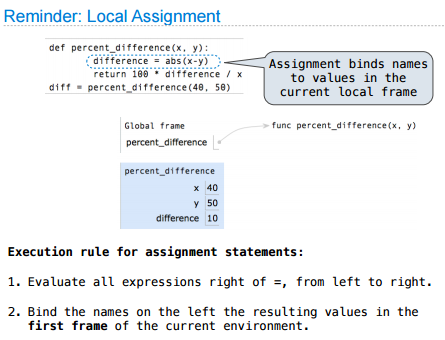


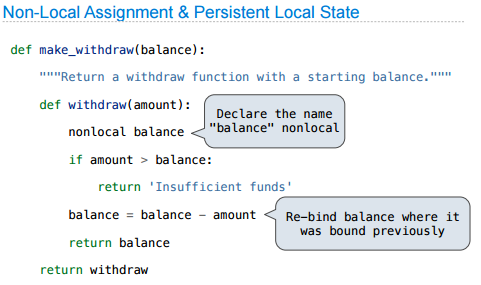


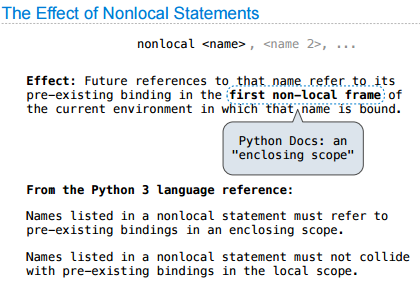


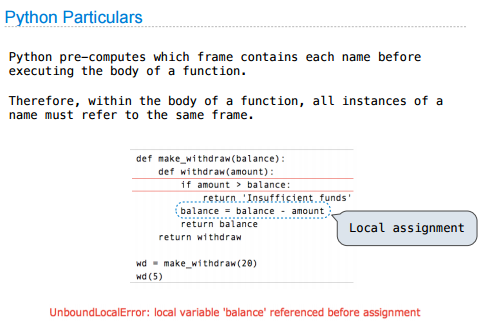


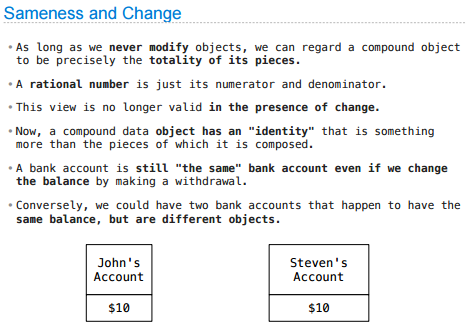


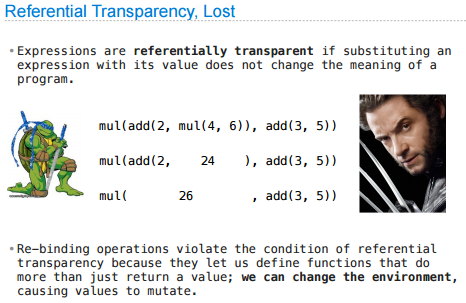


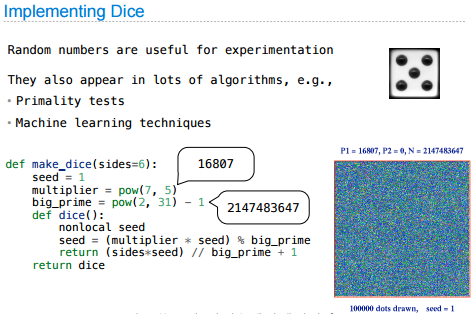


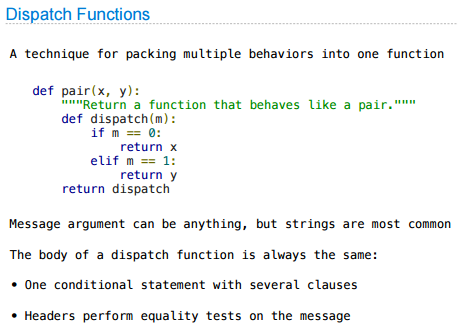


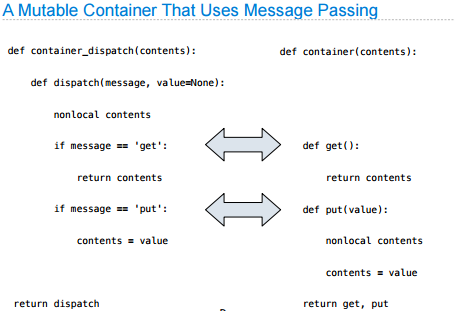


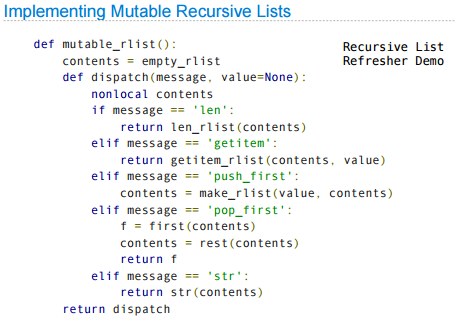


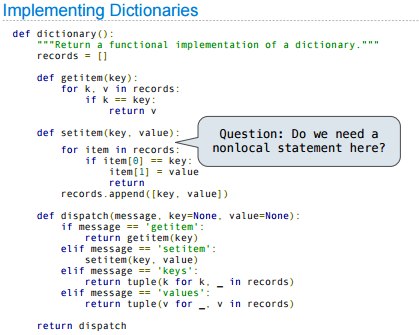


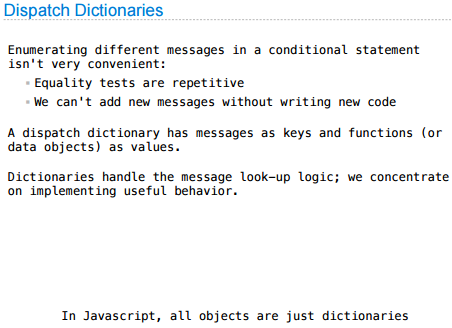












**Object oriented programming**

One powerful design strategy, which is particularly appropriate to the construction of programs for

modeling physical systems, is to base the structure of our programs on the structure of the system

being modeled. For each object in the system, we construct a corresponding computational object.

Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

