Motivation

• Important to know underpinnings of a programming language
  - Faster debugging:
    - Helps avoid weird bugs in your code
  - More efficient code:
    - Make better use of datatypes, commands
  - Maintainable by others
    - Code will be more consistent with others’ code

• Python:
  - How data is represented
  - How data is passed to functions

Quiz: How much do you know about Python

- What is the output?
  a = 5      a = [5]      for i in range(10):  for i in range(10):
  b = a      b = a          print(i)         print(i)
  a += 2      a += [2]      print(i)         i += 1
  print(a)    print(a)      print(a)         print(b)
  print(b)

- Can you write a function that increments variable passed to it?
  a = 5
  incr(a)
  print(a) => outputs 6

- Interaction with functions
  def addit(a,b):
    a += b
    return a
  a = 5
  c = addit(a,3)   a = [5]
  c = addit(a,[3])
  print(a)         print(a)
  print(c)         print(c)
Objects versus Variables

- Objects versus Variables
  - Objects are numbers, strings, lists, tuples, dictionaries, custom objects
  - This is a different view from other programming languages
  - Code might say how to create object
    + [1,2,3,4]
    + 5 + 2
  - Variables are used to refer to them
- C: very different
  - Some variables are pointers
  - But some variables actually store values (like ints, floats, boolean)
Mutable Objects

• Not talking about variables
• Mutable objects can be changed, or altered

• Example: List
  - a[2] = 5 ⇒ Can change individual elements
  - a.append(5) ⇒ Adds element onto end of list
  - a.extend([1,2]) ⇒ Adds 1 and 2 onto end of list
  - a += [1,2] ⇒ Same as extend
  - a.pop() ⇒ remove last element
  - a.sort() ⇒ changes order of list
• But not all operations on mutable objects change them
  - b=a[5] ⇒ a does not change
  - c=a+[1,2] ⇒ a does not change
  - sorted(a) ⇒ a does not change
  - c = a.copy() ⇒ a does not change
  - cnt = a.index(2) ⇒ a does not change

Immutable Objects

• Cannot be changed
  - There are no methods that allow object to be changed
• Numbers:
  - 5 is always 5
  - x=5; x+=2
    + 5 didn’t change. Instead, x now points to a different object
    + Points to object created from x+2
  - operator += works differently for lists versus strings
    + var += ...: lists: changes the object referenced by var
    + var += ...: numbers: creates a new object
Other Immutables

• Tuple
  - Cannot change the second element of (5,2)
  - Cannot extend it
  - Can use it in creating a new object, but that is not changing it

• Strings
  - s='abcde'; s+=f'
    + Not changing original string. Short form for s = s + f
    + Similar to a = 5; a += 2
    + To show immutable: s='abcde'; t=s; s+=f'; print(t)
  - Cannot do: s = 'abcde'; s[3] = 'D'
  - .strip(), .lower() etc create a new object

Why are Strings Immutable?

• Why did designers of python make strings immutable?
  - Why can’t we do s[3] = 'D'
  - This was a design decision

• My thoughts:
  - Immutable objects have fewer side effects
    a='the'; b=a; a+='m' ⇒ b has not changed
  - Be as consistent with other programming languages as possible
    + numbers, strings work like they do in other programming languages
Why are Lists Mutable?

• Why have any immutable objects at all?
  - Just use mutable objects for large objects where you don’t want to keep making copies to change things
  - Lists can sometimes be really long
  - If you have to make a new list to just change one element, going to be really slow
  - For smaller lists that don’t change, can just use a tuple
    + a tuple is an immutable list

• Consistent with C
  - You tend to use pointers for lists, especially to pass them to a function

Other Mutables

• Dictionary
• Sets
• Most programmer defined classes
Overview

- Objects
  ⇒ Equal
- Function Calls
- Implications

Equal Operator in Python

- \textit{var} = ‘\textit{instructions for creating an object}’:
  + object is created, and var is made to point to it
- \textit{var} = \textit{var1}: whatever \textit{var1} points to, so should \textit{var}
  - If \textit{var} pointed to something before, it no longer
  - If no one points to that object, it is garbage-collected
- Different view from many other programming languages
  - Python: everything is an object, variables point to an object
  - Other languages allow vars to also hold things, like a float, or an integer
    + Assignment copies the value
  - Everything is an object: this is semantics of python
    + Under the hood, python might treat numbers like other languages
Equal and Immutable Objects

- $\text{var} = \text{immutable object}$
  - immutable object can’t be changed so var will have same value until var points to something else
    + Example:
      $$\text{var} = 5$$
      $$\text{var} = \text{var} + 2$$
  - $\text{var} = \text{var1}$ where var1 points to an immutable
    + do not need to worry about side effects of var changing object
  - For immutables, same as if variable has its own copy of the value, rather than being a pointer
    + For $\text{var} = \text{var1}$, same as if we just copied the value of var1 of var

Equal and Mutable Objects

- $\text{var} = \text{mutable object}$
  - If some other variable also refers to the object, it can do an operation that changes the object, and so var’s value also changes
  - Can be side-effects

  a = [1, 2, 3, 4, 5]
  b = a
  a += [6]
  print(b)
Function Calls and Parameters

• Passing parameters really just equating it to a local var
  - And everything that that entails
    
    ```python
    def foe(d,e):
        d.extend(e)
    
    a = [1,2,3,4]  
    b = [5,6,7]    
    foe(a,b)       
    d = a; e = b  
    d.extend(e)   
    ```

• Local variable will be equated to the parameter
  - Python is sometimes referred to as: *Call by assignment*
Mutable versus Immutable Objects

• Function will change the object
  - only if object can be changed (mutable)
  - and only if you use a method that changes the object rather than creates a new object

• Passing pointer to a Mutable object
  - Seems like passing by reference

• Passing pointer to an Immutable object
  - Object cannot be changed
  - Seems like passing by value

Take-away

• Unclear if Python’s view is better than other views

• But, once you start doing any complex code, need to understand Python’s view

https://nedbatchelder.com/text/names1.html
Overview

- Objects
- Equal
- Function Calls
  ⇒ Implications

---

For

- On each iteration in loop, \( i \) gets assign to the next instance
  - If you change \( i \), does not affect next iteration in loop
    ```python
    for i in range(10):
        i += 1
        print(i)
    ```

- Tcl has two types of ‘for’, one traditional, and one for lists
  ```python
  for {set i 0} {i < 10} {incr i} {
      incr i
      puts i
  }
  foreach i in [0,1,2,3,4,5,6,7,8,9]:
      incr i
      puts i
  ```
Array initialization

```python
a = [0,0,0,0]
b = [a for i in range(10)]
b[9][0] = 5
print(b[3][0])
```

- `a` points to an array. `b` now has 10 pointers to that one array
- Proper way to do this:
  ```python
  b = [[0 for j in range(4)] for i in range(10)]
  ```
- Each outer iteration, a new subarray is created
  - Inner loop called once for each `i` in `range(10)`
  - Python does not create objects in advance, but at runtime

Is

- `is`:
  - checks whether same instance of object
  - not whether objects have same value
- There is just one True, False, None
- But is there just one 1?
  ```python
  a = 1
  b = 1
  if a == b:
    print('just one')
  ```
- Is there just one instance of all numbers?
  ```python
  for a,b in zip(range(500),range(500)):
    print(a==b)
    print(a is b)
  ```
Default Value for Function

```python
def foe(a=[]):
    a.append(1)
    return a
print(foe([1,2]))
print(foe())
print(foe())
```

- Function declarations are interpreted just once, when you load the file and default are assigned at that point
  - Makes sense since just want to interpret the declaration once, not everytime you run it
  - Default value is created at that time: just once, not each call
  - So if default value is a mutable, better not change it the function