Overview

Motivation
- Important to know underpinnings of a programming language
- How data is passed to functions
- How data is represented

Python
- How data is represented
- How data is passed to functions
- More efficient code:
  + Make better use of datatypes, commands
  + Helps avoid weird bugs in your code
  + Faster debugging
- Maintainable by others:
  + Code will be more consistent with others code
  + Will help others understand and use code
- Faster debugging:

Inequalities
- Function Calls
- Equal
- Objects
Mutuable Objects

• Not talking about variables
• Mutuable objects can be changed, or altered
  - Example: List
    + `a[2] = 5` ⇒ Can change individual elements
    + `a.extend([1,2])` ⇒ Can extend the list
    + `a += [1,2]` ⇒ Can extend the list
    + `a.append(5)` ⇒ Adds element onto the end
    + `a.pop()` ⇒ Remove last element
    + `a.sort()` ⇒ Changes order of list

• But not all operations mutable objects change them
  - `b = a[5]` different from `a + [1,2]`
  - `c = a + [1,2]` different from `a += [1,2]`}

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
  - Variables are numbers, strings, lists, tuples, dictionaries, custom objects
  - Some variables are pointers
  - Variables are used to refer to them
  - Very different

Not talking about variables

Mutuable Objects
Other Immutables

- Tuple
  - Cannot change the second element of (5,2)
  - Cannot extend it
  - Can use it in creating a new object

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

Immutable Objects

- Cannot be changed
- There are no methods that allow object to be changed
- Cannot be changed

- Other
- Cannot be changed
- Cannot be changed

- Numbers
  - Points a new object
  - Works differently for lists versus strings
  - Points to object created from a+2
  - 5 is always 5

- Tuple
  - Cannot change the second element of (5,2)

- Strings
  - Cannot change the second element of (5,2)
  - Cannot extend it
  - Can use it in creating a new object

- Other
  - Cannot be changed
  - Cannot be changed

- Numbers
  - Points a new object
  - Works differently for lists versus strings
  - Points to object created from a+2
  - 5 is always 5

- Tuple
  - Cannot change the second element of (5,2)

- Strings
  - Cannot change the second element of (5,2)
  - Cannot extend it
  - Can use it in creating a new object
**Why are Strings Immutable?**

- Why didn't we do s[3] = 'D'?

  - Why can't we make strings immutable?
  - Why did designers of Python make strings immutable?

  - Immutable objects have fewer side effects.
  - Just use mutable objects for large objects where you don't want to keep making copies to change things.
  - Be as consistent with other programming languages as possible.
  - Numbers, strings work like they do in other programming languages.

**Other Mutuables**

- Sels
- Dictionary

- Most programmer defined classes
Equal Operator in Python

Overview

• Objects
⇒ Equal

• Function Calls

• Implications

• Function Calls

• Equal
⇒ Objects

Different view from many other programming languages.

- Python: Everything is an object, variables point to objects.
- Other languages allow variables to also hold things like a float, or an integer.
- Python: Assignment copies the value.
- Other languages allow assignment to also hold things like a float, or an integer.
- Python: Everything is an object, variables point to objects.

Under the hood, Python treats numbers like other languages.

If no one points to that object, it is garbage-collected.

If var points to something before, then var is no longer.

var = var1: Whatever var1 points to, so should var.

var = instructions for creating an object:

⇒ var is garbage-collected and var is made to point to it.
Equal and Immutable Objects

- var = immutable object
- immutable object can't be changed so var will have same value until var points to something else
- For immutable, same as if variable has its own copy of the value, rather than being a pointer
- For immutable, same as if variable has its own copy of the value, rather than being a pointer
- do not need to worry about side effects of var changing object
- var = var1
- where var1 points to immutable
- var = var
- var = 2
- var = 3
- Example:
- Points to something else
- Immutable object can't be changed so var will have same value until var
- var = immutable object

Equal and Mutuable Objects

- a = var
- a += [6]
- a = [1, 2, 3, 4, 5]
- var = mutable object
- Can be side-effects
- Changes the object, and so var's value also changes
- If some other variable also refers to the object, it can do an operation that
- var = mutable object
Function Calls and Parameters

- Passing parameters is really just equating parameter to a local variable - And everything that entails

```python
def foe(d, e):
    a = [1, 2, 3, 4]
    d.extend(e)
    b = [5, 6, 7]
    d = a
e = b
d.extend(e)
a = [1, 2, 3, 4]
b = [5, 6, 7]
d.extend(e)
foe(a, b)
```

- Local variable will be equated to the parameter Python is sometimes referred to as: Call by assignment
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

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
  - only if object can be changed (mutable)
  - Function will change the object

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

• Passing pointer to an Immutable object - seems like passing by reference
  - new object

Passing pointer to a Mutable object - seems like passing by reference
  - new object

- Seams like passing by value
  - Object cannot be changed

On each iteration in loop, i gets assigned to the next instance. If you change i, does not affect next iteration in loop.

TL;DR has two types of 'for', one traditional, and one for lists.

```python
for i in range(10):
    i += 1
    print(i)
```

```tcl
for {set i 0} {i < 10} {incr i} {
    incr i
    puts i
}
```

```tcl
foreach i in [0,1,2,3,4,5,6,7,8,9]:
    incr i
    puts i
```
is:

```
if a == b:
    print('just one')
```

```
for a,b in zip(range(500),range(500)):
    print(a==b)
    print(a is b)
```

Array initialization

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

```
b = [[0 for j in range(5)] for i in range(10)]
b[9][0] = 5
print(b)
```

Python does not create objects in advance: but at runtime
- Each outer iteration of b creates a new subarray
  - Inner loop called once for each i in range(10)
```
b = [0][6] # b = [0][0][0][0][0][0][0][0][0][0]
print(b)
```

A property to an array, q now has 10 pointers to that one array
Default Value for Function

- Default value is created at that time; just once, not each call
- Makes sense since just want to interpret the declaration once, not the life and default are assigned at that point

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

- Function declarations are interpreted just once, when you load
- Python may or may not represent two objects with same value as same object

Strings?

```python
a = '12345'
b = '12345'
if a is b:
    print('just one')
```

- Does compiler make one of two objects?
- Not guaranteed

Strings?

```python
if a is b:
    print('just one')
```

- Python may or may not represent two objects with same value as same object

- Function declarations are interpreted just once, when you load

- Default value is created at that time; just once, not each call