Workshop 1

Object Types

Object Type	Example literals / creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	‘spam’, “bob’s”, b’a\x01c’, u’sp\xc4m’
Lists	[1, [2, ‘three’], 4, 5], list(range(10))
Dictionaries	{‘food’: ‘spam’, ‘taste’: ‘yum’}, dict(hours=10)
Tuples	(1, ‘spam’, 4 ‘U’), tuple(‘spam’), namedtuple
Files	Open(‘eggs.txt’), open(r’C:\ham.bin’, ‘wb’)
Sets	Set(‘abc’), {‘a’, ‘b’, ‘c’}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

Python is dynamically typed: Python types the object for you. There is no type definition statement in Python.

Python is strongly typed: you can perform on an object only the operations that are valid for that object type.

Numeric Data Types

Integer (3, 42) and floating-point (3.141592, -0.345) objects

Decimal: fixed precision objects (Decimal(‘1.3’))

Fraction: rational number objects (Fraction(1, 3))

Sets: collections with numeric operations (set(‘spam’), {1,2,3,4})

Booleans: true and false (also 1 and 0)

Built-in functions and modules (round, math, random)

Expressions: unlimited integer precision

Operators: +, -, *, /, **

Precedence – what you would expect

Mixed types: simple types are converted up to more complex types in the same expression

```
>>>1/3                    # Python 2.7
0

>>>1.0/3
0.33333333333333
```

Floating point numbers: there is likely to be some residual amount in the fractional part that will cause the following:

```
>>> 1.1 + 2.2 == 3.3
False
>>> 1.1 + 2.2
3.3000000000000003
>>>
```

Comparisons

<, >, == , !=

```
>>>2 < 3
True

>>>2 != 3
True

>>>2 > 3
False
```

Decimals

The decimal type is used for fixed precision objects. For example, accounting applications might want exactly two digits to the right of the decimal. We have this object type because numbers that have a decimal part aren't always represented exactly as expected in a computer (we just saw that above):

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17

>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Fractions

Fractions work much the same way as Decimals.

```
>>> from fractions import Fraction
>>> q = Fraction(1,3)
>>>q + q
Fraction (2, 3)
>>>r = Fraction(1,6)
>>>r + q
Fraction(1,2)
```

Sets

Note that sets are unordered and immutable. The set object implements the mathematical concept of a set.

Operations

Difference (-), Union (|), Intersection (&), Superset (>), Subset (<), Symmetric difference or XOR (^)

Symmetric difference returns the elements that do not appear in both sets.

To create an empty set:

```
>>> S = set()
```

More set examples:

```
>>> S = {1, 2, 3, 'a', 'b'}
>>>R = {2, 4, 'b', 'd'}
>>> S | R
{1, 2, 3, 4, 'd', 'b', 'a'}
>>>S&R
{'b', 2}
>>>S-R
{1, 3, 'a'}
```

Sets are good for eliminating duplicates, isolating differences, order-neutral equality (same elements but not necessarily in the same order), finding common elements in two groups (intersection).

The set operations will often be more efficient than the corresponding loop operation to examine each item in a group.

Strings

A string is a Python *sequence* – a positionally ordered collection of objects.

‘spam’

‘This is a string.’

Python indexes times by their offset from the front of the sequence. The first item in a sequence is item zero. (This is also known as “zero-based addressing”.)

```
>>>S = ‘spam’
>>>S[0]
‘s’
>>>S[1]
‘p’
```

A negative index is an offset from the end of the sequence:

```
>>>S[-1]
‘m’
>>>S[-2]
‘a’
```

Slicing: Slicing gives everything from the first index up to but not including the last index.

```
>>>S=’spam’
>>>S[1:3]
‘pa’
```

Lists

Lists are mutable sequences. All of the ways of working with indexes in strings applies to lists. Since lists are mutable, you can also change an element in the list.

```
>>>L = [123, ‘spam’, 1.23]
>>>len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, ‘spam’]
>>> L + [4, 5, 6]
[123, ‘spam’, 1.23, 4, 5, 6]
>>> L * 2
```

```
[123, 'spam', 1.23, 123, 'spam', 1.23]
>>> L
[123, 'spam', 1.23]
```

Since lists are mutable, you can change an item in a list:

```
>>> L[1] = 'eggs'
[123, 'eggs', 1.23]
```

```
>>> L[99]
```

Generates an error.

Nesting

```
>>> M = [[1,2,3], [4,5,6], [7,8,9]]
>>> M
[[1,2,3], [4,5,6], [7,8,9]]
```

```
>>> M[1]
[4, 5, 6]
```

```
>>>M [1][2]
6
```

Dictionaries

Dictionaries are <u>not</u> sequences. Dictionaries are mappings. The primary difference between a list and a dictionary is that items in a list appear in a sequence whereas items in a dictionary are located by a key value. Dictionaries are mutable.

```
>>>D = {'food':'spam', 'qty':4, 'color':'pink'}
>>>D['food']
'spam'
```

Items in a dictionary can be complex.

```
>>>D = {'food': ['lettuce', 'carrots'], 'qty': [4, 12], 'status': ['fresh', 'fresh']}
>>>D['food']
['lettuce', 'carrots']
```

```
>>>[f for f in D['food']]
['lettuce', 'carrots']
```

```
>>>[f + ' ok' for f in D['food']]
['lettuce ok', 'carrots ok']
```

```
>>>D['name']
```

Generates an error

Dictionary keys may occur in any order, so if you need the data in the list in order by key value, put the keys in a list, sort the list, and use the sorted list to access the data. Note that this data will be sorted by key value, not data value.

Tuples

Tuples are sequences like lists, but immutable like strings.

```
>>>T = (1, 2, 3, 4)
>>>T[1:]
(2, 3, 4)
```

A tuple is a good choice if you need an object that looks like a list but you don't want it to be changed. For example, you might need a list of values that are associated with states (say, population in the last census by state).

Files, Classes, and OOP

We won't get into these topics in depth in this class. You will see examples of how to open files and write to them. Classes and OOP we won't address.

Mutable versus Immutable

A key concept is immutable vs. mutable. Numbers, strings, and tuples are immutable: they cannot be changed. In other words, 2 is always 2, 'spam' is always 'spam', and (1, 'spam', 4 'U') is always (1, 'spam', 4 'U').

You cannot change 'spam' to 'scam' by doing something like:

```
>>>S='spam'
>>>S[1] = 'c'
```

This will generate an error message.

However, the following is OK because it creates a new string:

```
>>>S = 'spam'
>>>S = 'z' + S[1:]
zpam
```

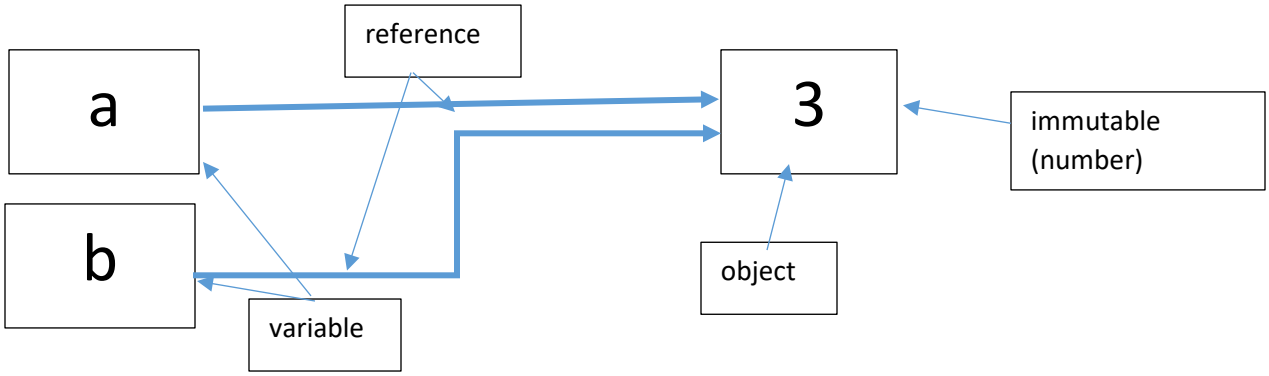
Dynamic Typing

Variables: created when your code first assigns a value.
Variables do not have a type. Objects have a type.
When used, variables are replaced with the object they reference.

>>> a = 3
Create an object 3.
Create a variable a (if it does not exist).
Link the variable a to the object 3. (This is a reference.)

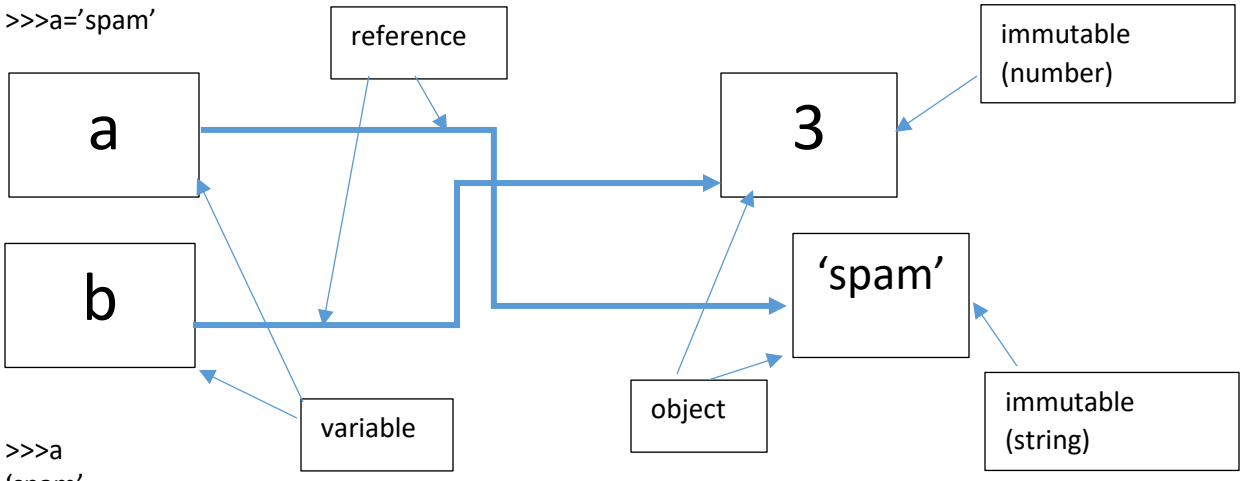
When dealing with an immutable object:

```
>>> a=3
>>> b=a
```




```
>>>a
3
>>>b
3
```

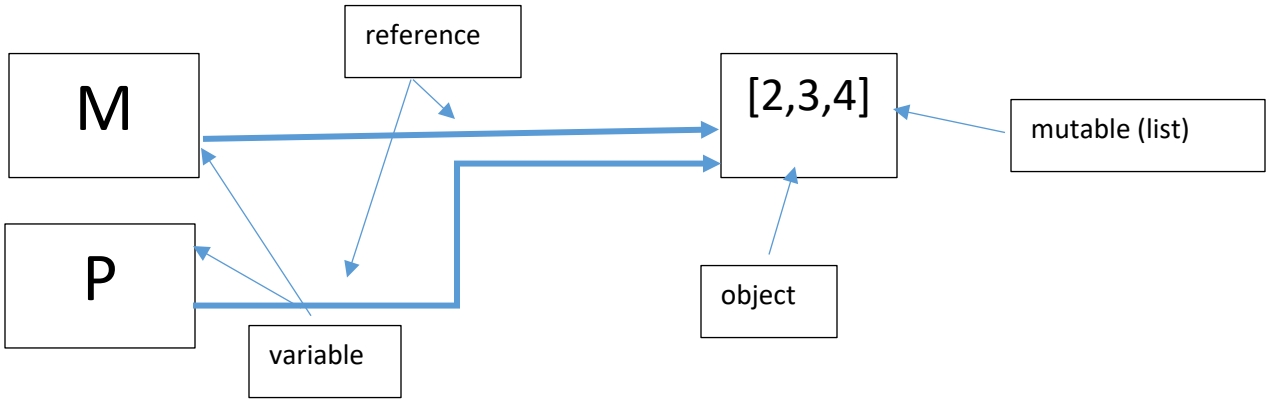
```
>>>a='spam'
```



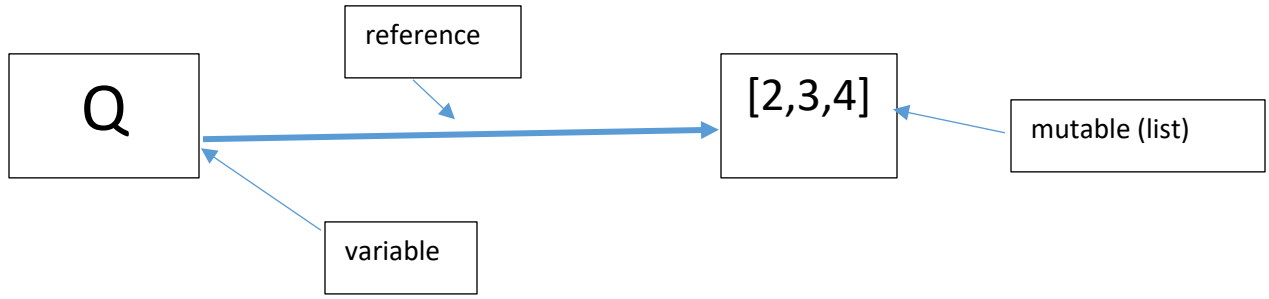

```
>>>a
'spam'
>>>b
3
```

But, when dealing with mutable objects:

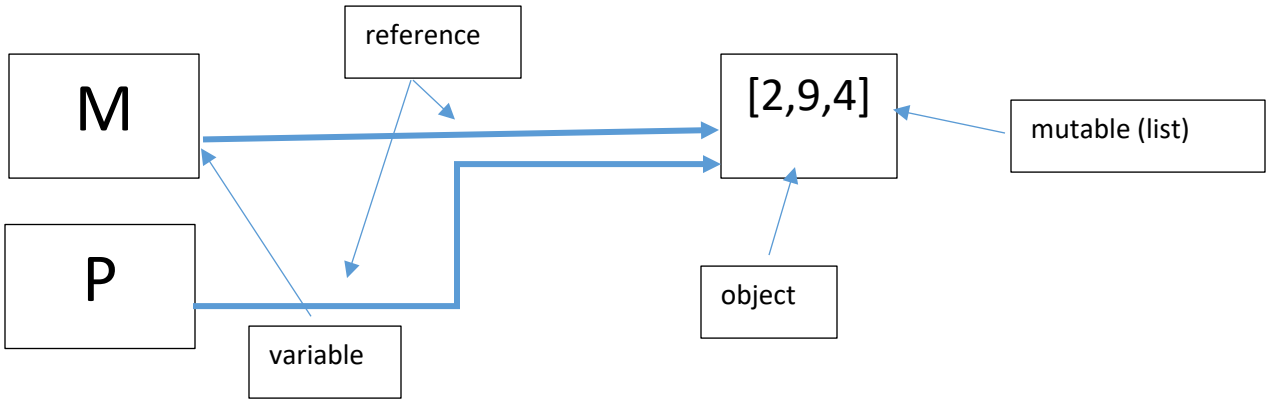
```
>>>M = [2,3,4]
>>>P = M
```




```
>>>Q = [2, 3, 4]
```




```
>>>M[1] = 9
```




```
>>>P
[2, 9, 4]
>>>Q
[2, 3, 4]
```

So, we have this situation:

```
>>>M = [1, 2, 3]
>>>P = M
>>>M == P              #Same values? Yes.
True
>>>P is M              #Same object? Yes. P references M.
True

>>>Q = [1, 2, 3]
>>>Q == M              #Same values? Yes.
True
>>>Q is M              #Same object? No. Lists are a mutable object, so a new one is created when
False                  #  Q is assigned a value.

>>>X = 42
>>>Y = 42
>>>X == Y              #Same value? Yes.
True
>>>X is Y              #Same object? YES. Numbers are immutable objects.
True
```

Let's write some code!

Write a program that outputs "Hello, world!" [This is a simple exercise to familiarize you with how to write code in a text file and then get you computer to run that code. Solution: print('Hello, world!')]

Write a program that will ask the user for three integers and output the average of the numbers.

```
num1 = int(input('Enter the first number:'))
num2 = int(input('Enter the second number:'))
num3 = int(input('Enter the third number:'))
avg = (num1 + num2 + num3) / 3.0
print(str(avg))
```

Change the int(…) to float(…) and rerun the program.

Change the program to ask for your name and print your output in a user friendly manner:

Code:

```
user = input('What is your name?')
…
print(user, 'the average is', str(avg))
```