# BINF5007 Lec 2

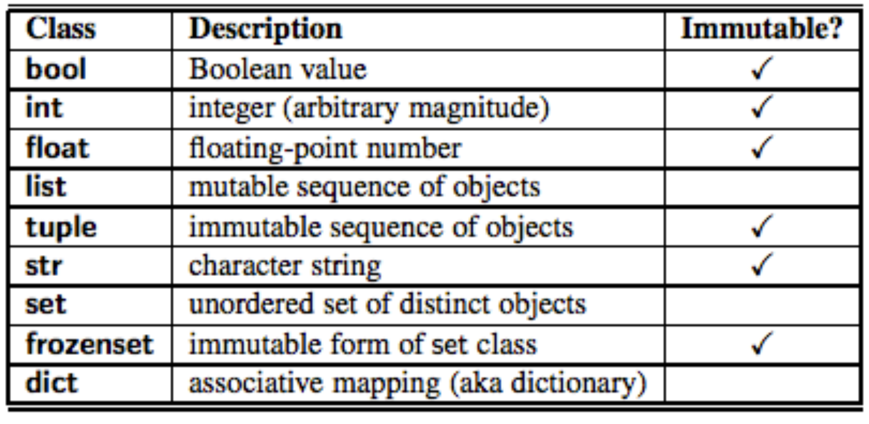
Python Data Structures

# Summary

- Mutable / Immutable
- Data Types
- Lists
- Built-on Functions and Lists
- Tuple

# Immutable vs Mutable

- An **immutable** variable cannot be changed after it is created (with one caveat)
- If you wish to change an **immutable** variable, such as a string, you must create a new instance and bind the variable to the new instance
- A **mutable** variable can be changed in place
- Refer to this list of the Python data types, and whether they are mutable
- We'll come back to this Lecture 2 for a deeper dive



| Class | Description | Immutable? |
|---|---|---|
| **bool** | Boolean value | ✓ |
| **int** | integer (arbitrary magnitude) | ✓ |
| **float** | floating-point number | ✓ |
| **list** | mutable sequence of objects | |
| **tuple** | immutable sequence of objects | ✓ |
| **str** | character string | ✓ |
| **set** | unordered set of distinct objects | |
| **frozenset** | immutable form of set class | ✓ |
| **dict** | associative mapping (aka dictionary) | |

# Types

# What Does "Type" Mean?

- In Python variables, literals, and constants have a "type"
- Python knows the difference between an integer number and a string
- Why's that important?
- So "+" means "addition" if something is a **number** and "concatenate" if something is a **string**

```
>>> sum = 1 + 4
>>> print(sum)
5
>>> str1 = 'hello ' + 'there'
>>> print(str1)
hello there
```

concatenate = put together

# Type Matters

- Python knows what "`type`" everything is
- But, some operations are prohibited
- You cannot "`+ 1`" to a string**
- We can ask Python what type something is by using the `type()` function

```
>>> str1 = 'hello ' + 'there'
>>> str1 = str1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> type(str1)
< ass'str'>
>>> type('hello')
< ass'str'>
>>> type(1)
< ass'int'>
>>>
```

Use **`print(type(variable))`** as your friend when coding!!

** Technically you could *Typecast* 1 into a string with **`str(1)`**

** But can you 1 + **`int("hello")`**? **Try it out with Python REPL**

# Several Types of Numbers

- Numbers have two main types
  - Integers are whole numbers: -14, -2, 0, 1, 100, 401233
  - Floating Point Numbers have decimal parts: -2.5 , 0.0, 98.6, 14.0
- There are other number types - they are variations on float and integer

```
>>> xx = 1
>>> type(xx)
< ass 'int'>
>>> temp = 98.6
>>> type(temp)
< ass'float'>
>>> type(1)
< ass 'int'>
>>> type(1.0)
< ass'float'>
>>>
```

# Why Do We Need Two Number Types?

- Operations on **ints** produce **ints**, operations on floats produce floats (except for /).

```
>>> 3.0 + 4.0
7.0
>>> 3 + 4
7
>>> 3.0 * 4.0
12.0
>>> 3 * 4
12
>>> 10 / 2
5.0
>>> 10.0 / 3.0
3.3333333333333335
>>> 10 / 3
3.3333333333333335
>>> 10 // 3 # see coming slides 3
>>> 10.0 // 3.0 # integer division 3.0
```

# Integer Division

Python 3.x Integer division produces a **floating point result**

This was different in Python 2.x

```
Python 3.6.6..
..
..
>>> print(10 / 2)
5.0
>>> print(9 / 2)
4.5
>>> print(99 / 100)
0.99
>>> print(10.0 / 2.0)
5.0
>>> print(99.0 / 100.0)
0.99
```

# Modulo Operator

The % operator (modulo or mod) finds the remainder of a division, and return an **`int`**

- Possible results are between 0 (in usive) and the right hand side operand (ex usive)  Example: for **`x % 3`**, the only results are `0`, `1`, or `2`
  - `6 % 3 → 0`
  - `7 % 3 → 1`
  - `8 % 3 → 2`
  - `9 % 3 → 0`
- Uses for modulo operator:
  - Even/odd: `n` is even `if n % 2 == zero   # good for Even / Odd and Alternating`
  - " ock arithmetic"
    - Minutes are mod 60: 3:**58** + 15 minutes = 4:**13**      **73 % 60  = 13**
    - Hours are mod 12:  10:00 + 4 hours = **2:00**      **14 % 12 = 2**

Updated Neil Moore

# Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float (implicit type conversion)
- Sometimes we want to control the type conversion
- You can control this with the built-in functions `int()` and `float()`
- This is called *explicit typing* (explicit type conversion)

```
>>> float(99) + 100
199.0
>>> i = 42
>>> type(i)
< ass'int'>
>>> f = float(i)
>>> f
42.0
>>> type(f)
< ass'float'>
>>> int(3.9)
3
>>> round(3.9)
4
>>> round(3.9, 1)
3.9
```

Upated

# More on Type Conversions

- What happens when you mix an **int** and **float** in an expression?
  `x = 5.0 + 2`
- What do you think should happen?
- For Python to evaluate this expression, it must either convert 5.0 to 5 and do an integer addition, or convert 2 to 2.0 and do a floating point addition
- But, converting a float to an int will lose information!
- Ints can easily be converted to floats by adding ".0"

# Rounding

Builtin function – do NOT have to import math library to use it

- `round` has **either *one*** or ***two*** arguments
  - ONE argument: round the argument to the **nearest integer**
    - `round(5.2)   # 5`
    - `round(7.9)   # 8`
  - TWO arguments, the second argument is the number of decimal places desired.  The first argument's value will be rounded to that number of decimals
    - `round(math.pi, 2) # 3.14`
    - `round(2.71818, 0) # 3.0`
    - `round(2.71818, 1) # 2.7`
    - `round(12, -1)        # 10`
    - `round(121, -2)    # 100`

Updated Neil Moore

# String Conversions

- You can also use `int()` and `float()` to convert between strings and integers

- But you will get an error if the string does not contain numeric characters

- Important to recognize the **Stack traces** and **Exceptions** (`TypeError` and `ValueError`)

```
>>> sval = '123'
>>> type(sval)
< ass 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object
to str implicitly
>>> ival = int(sval)
>>> type(ival)
< ass 'int'>
>>> print(ival + 1)
124
>>> str1 = 'hello bob'
>>> niv = int(str1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'x'
```

# Python Lists

# What is Not a "Collection"?

Some of the variables ***so far*** have had one value in them (int, float, string) - when we assign the object to a new variable, the old label *is changed to point to the new object*

```
$ python
>>> x = 2
>>> x = 4
>>> print(x) 4
```

# A List is a Kind of Collection

- A collection allows us to put many values in a single "variable"
- A collection is nice because we can carry many values around in one convenient package

```
friends = ['John', 'Glenn', 'Sally']

carryon = ['socks', 'shirt', 'perfume']
```

# List Constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas
- A list element can be any Python object - even another list
- A list can be empty

```
>>> print([1, 24, 76])
[1, 24, 76]

>>> print(['red', 'yellow', 'blue'])

['red', 'yellow', 'blue']

>>> print(type(['red', 24, 98.6]))
< ass 'list'>

>>> print([ 1, [5, 6], 7])
[1, [5, 6], 7]

>>> print([])
[]
```

# Looking Inside Lists

We can get at any single element in a list using an index specified in square brackets

|  |  |  |
|---|---|---|
| Joseph | Glenn | Sally |
| 0 | 1 | 2 |

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print(friends[1])
Glenn
```

# Lists are Mutable

- Strings are "immutable" - we cannot change the contents of a string - we must make a new string to make any change
- But Lists are "mutable" - we can change an element of a list using the index operator

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
Traceback
TypeError: 'str' object does not
support item assignment

>>> fruit = fruit.lower()
>>> print(fruit)
banana
```

# Concatenating Lists Using +

We can create a new list by adding two existing lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```

# Break

# Lists Can Be Sliced Using :

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]

[41, 12]
>>> t[:4]
[9, 41, 12,
            3]
>>> t[3:]
[3, 74, 15]


>>> t[:]
[9, 41, 12, 3, 74, 15]
>>> id(t)
140305575894528
>>> s = t[:]
>>> id(s)
140305575894592  # what happened here?  It's a way to create a copy
```

Remember: the second number is "up to but not in uding"

Same behavior in strings

# Building a List from Scratch

- We can create an empty list and then add elements using the append method

- The list stays in order and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print(stuff)
['book', 99]
>>> stuff.append('cookie')
>>> print(stuff)
['book', 99, 'cookie']
```

# Is There a Particular Value in a List?

- Python provides two operators that let you check if an item is in a list
- These are logical operators that return `True` or `False`
- They do not modify the list

```
>>> some = [1, 9, 21, 10, 16]
>>> 9 in some
True
>>> 15 in some
False
>>> 20 not in some
True
```

# Built-in Functions and Lists
# What's the Length of a List?

- The `len()` function can take a list as a parameter and returns the number of elements in the list

- Actually `len()` tells us the number of elements of any **set** or **sequence** (such as a string...)

```
>>> greet = 'Hello Bob'
>>> print(len(greet))
9

>>> x = [ 1, 2, 'joe', 99]
>>> print(len(x))
4
```

# Built-in Functions and Lists
# Using the range Function

- The `range()` function returns *range* of numbers that range from zero to **one less** than the parameter. To get a list, you just call **`list()`**

- The `range()` function is used to generate a sequence of numbers over time. At its simplest, it accepts an integer and returns a `range` object (a type of **iterable**)

- We'll come back to some of the nuances of iterables and `range()` later on this semester

```
>>> print(range(4))
range(0, 4)

>>> print(list(range(4)))
[0, 1, 2, 3]

>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print(len(friends))
3

>>> print(list(range(len(friends))))
[0, 1, 2]
```

# Additional Built-in Functions and Lists

- There are a number of other functions built into Python that take lists as parameters
- In other languages we might need to use loops, but these are much simpler to calculate in Python (although knowing the loop pattern is good, as we'll see later on)
- Technically, these functions will work on iterables

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(max(nums))

>>> print(min(nums))

>>> print(sum(nums))

>>> print(sum(nums)/len(nums))
```

# Additional Built-in Functions and Lists
# Sorting a List (ASCII)

- A list can hold many items and keeps those items in the order until we do something to change the order

- A list can be sorted (i.e., change its order)

- The sort method means "sort yourself" in place. i.e. you use it **w/o assignment**

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> friends.sort()
>>> print(friends)
['Glenn', 'Joseph', 'Sally']
>>> print(friends[1])
Joseph
>>>
```

# Strings and Lists

```
>>> str1 = 'With three words'
>>> stuff = str1.split()
>>> print(stuff)
['With', 'three', 'words']
>>> print(len(stuff))
3
>>> print(stuff[0])
With
```

```
>>> print(stuff)
['With', 'three', 'words']
>>> for w in stuff :
...     print(w)
...
With
Three
Words
>>>
```

`split` breaks a string into parts and produces a list of strings.  We think of these as words. We can access a particular word or loop through all the words.

```
>>> line = 'A lot             of  spaces'
>>> etc = line.split()
>>> print(etc)
['A', 'lot', 'of', 'spaces']

>>> line = 'first;second;third'
>>> thing = line.split()
>>> print(thing)
['first;second;third']
>>> print(len(thing))
1

>>> thing = line.split(';')
>>> print(thing)
['first', 'second', 'third']
>>> print(len(thing))
3
```

- When you do not specify a delimiter, multiple spaces are treated like one delimiter

- You can specify what delimiter character to use in the splitting

# Tuples

# Tuples Are Like Lists

Tuples are another kind of sequence that functions much like a list - they have elements which are indexed starting at 0

```
>>> x = ('Glenn', 'Sally', 'Joseph')
>>> print(x[2])
Joseph
>>> y = (1, 9, 2)
>>> print(y)
(1, 9, 2)
>>> print(max(y))
9
```

# but... Tuples are “immutable”

Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
>>>[9, 8, 6]
>>>
```

List

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback:'str'
object does
not support item
Assignment
>>>
```

String

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple'
object does
not support item
Assignment
>>>
```

Tuple

# A Tale of Two Sequences

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

# Tuples and Assignment

- We can also put a tuple on the left-hand side of an assignment statement
- We can even omit the parentheses

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred

>>> a, b = (99, 98)
>>> print(a)
99
```

# Why have tuples?

- Tuples are More Efficient
- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- So, in our program when we are making “temporary variables” we prefer tuples over lists
- But there's more to it as well…

# Why not just use a list instead?

- Lists should hold a variable quantity of objects
- E.g. list containing the file names of all the files in a directory can be stored in a list
  - All of the same type (string), and the number of elements in the list changes according to each directory
  - The element ordering inside this list is not relevant
- A List makes sense here…

# Using a Tuple

- A typical example of a tuple is a coordinate system
- In a 3-D coordinate system, each point is referred to by a three element tuple `(x, y, z)`
- The number of elements for each tuple does not change (since there are always three coordinates), and each position is important since each point corresponds to a specific axis
- We can say the same thing regarding the elements that are returned from a function or a dictionary key

# Packing / Unpacking Tuples

- One way to think of tuple assignment is as tuple packing/unpacking
- In tuple packing, the values on the left are ‘packed’ together in a tuple:

```
>>> b = ("Bob", 19, "CS")     # tuple packing
```

- In tuple unpacking, the values in a tuple on the right are ‘unpacked’ into the variables/names on the right:

```
>>> b = ("Bob", 19, "CS")
>>> name, age, studies = b     # tuple unpacking
>>> studies
'CS'
```

- ***Remember***, Tuple assignment solves swapping values neatly

```
(a, b) = (b, a)
```

https://openbookproject.net/thinkcs/python/english3e/tuples.html

# When To Use Tuples over Lists

- Well, obviously this depends on your needs and use cases
- There may be some occasions you specifically do not what data to be changed
- If you have data which is not meant to be changed in the first place, you should choose tuple data type over lists
- But if you know that the data will grow and shrink during the runtime of the application, you need to go with the list data type

https://towardsdatascience.com/python-tuples-when-to-use-them-over-lists-75e443f9dcd7

# Key Takeaways and Confusion

- The key difference between tuples and lists: tuples are **immutable** objects the lists are **mutable**
- Tuples are more memory efficient than the lists
- When it comes to the time efficiency, again tuples have a slight advantage over the lists especially when lookup to a value is considered
- If you have data which is not meant to be changed in the first place, you should choose the tuple data type over lists

https://towardsdatascience.com/python-tuples-when-to-use-them-over-lists-75e443f9dcd7