ATMS 305 WEEK 4: INTRODUCTION TO PYTHON

Lecture 2: Objects, attributes, and methods

COMPLEX NUMBERS: AN EXAMPLE OBJECT!

Python can represent complex numbers with the following syntax:

```
a = 4.9 - 8.3j
b = complex(5.3,2.4)
```

It turns out that `a` is an object (of complex type)

OBJECTS HAVE ATTRIBUTES

```
a = 4.9 - 8.3j
a.real
4.9
a.imag
-8.3
```

Objects (and methods, see next slide) can be identified in ipython by typing tab after typing the variable name followed by a period (.).

Note that there are no parentheses associated with attributes.

METHODS

Methods are built in functions that are attached to an object.

In the case of complex number objects, the method conjugate will calculate the complex conjugate of a number.

You can tell the difference between a method and and object because it requires parentheses, even if it does not require keywords.

```
a.conjugate
<function conjugate>
a.conjugate()
(4.5-6.4j)
```

```
b = a.conjugate()
b()
```

Methods can be stored as variables to be called later! How fun! 😎😎😎

RUNNING ON EMPTY: NONE

None is the null set, or an empty variable.

It is not the same as 0.

```
a = None #an empty variable
b = [None]  #an empty list
```

COLLECTING INFORMATION: LISTS AND TUPLES

So far we have focused on single value variables.

python has many built in ways to collect values or objects.

Lists are collections, enclosed by square brackets, that are mutable.

Tuples are collections, enclosed by parentheses, that are immutable.

Mutable is a fancy way of saying that the contents can be changed after creation.

Immutable…well you can figure it out 🤗

```
l = [4.5, -7.8, 'pickle', True, None, 5]  # list
t = (4.5, -7.8, 'pickle', True, None, 5)  # tuple
```

ACCESSING INFORMATION IN LISTS AND TUPLES

Let's say we created a list

```
mylist = [0, -5, 8, 'hi', (1,3,5), False]
```

We can access individual elements in a list using integer subscripts enclosed in brackets.

```
mylist[1]
```

```
-5
```

Try

```
mylist[-2]
```

NESTED ELEMENTS

Sometimes we may want to contain lists or tuples within a list or tuple.

Example:

```
mylist = [2, [4, 5, 6], [5]]
```

Try:

```
mylist[1]
mylist[1][1]
mylist[1][-1]
```

RANGES OF ELEMENTS

One way to select a range of elements in a list or tuple is to use the colon between two integers:

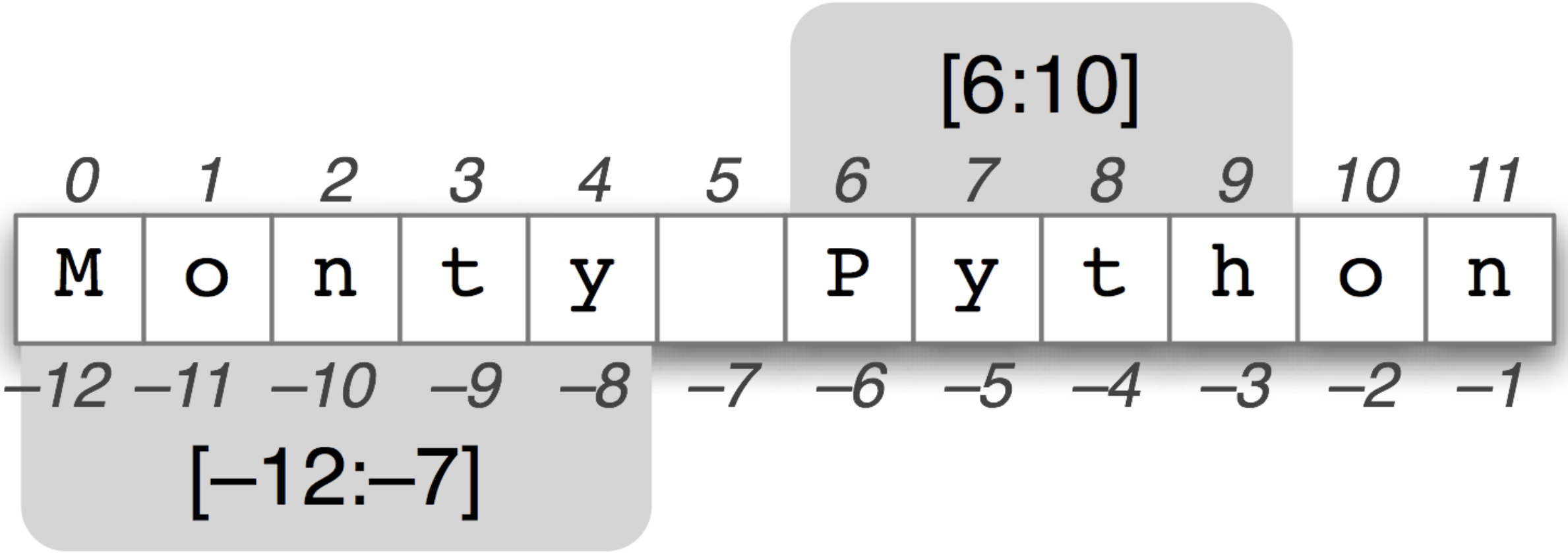
```
mylist2 = [6, 4.5, 8, 'jerky', [3, 6, 2]]
```

Try:

```
mylist2[1:3]
mylist[2:-1]
mylist[3:5]
```

Remember that python is 0 based, and ranges do not include the index at the end of the range.

INDEX ILLUSTRATION



OTHER INDEXING TIPS AND TRICKS

```
mylist = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

`[:]` get all the values

`[2:]` get from the third index onward

`[:-2]` get from the first to the second to last index

`[0:-1,2]` get every other index

`[::-1]` get the indices in reverse order

`[-1,0,-2]`

`[::-2]`

REASSIGNING ELEMENTS OF LISTS ~~AND TUPLES~~ 😱

You can reassign individual values or ranges of a list with other elements.

You can't do that with a tuple, because tuples are **immutable**.

```
mylist = ['raiders','chargers','49ers','rams']
mylist[3] = 'seahawks'
mylist[0:1] = ['saints','chiefs']
mylist[0:1] = 'texans'
```

HOME, HOME ON THE RANGE 🤠

the range() function will return a list of numbers for you

```
range(3,5)
range(10)
range(10,-1,3)
```

OTHER FUNCTIONS AND METHODS FOR LISTS

`len(mylist)`	return length of list
`del mylist[a:b]`	delete list elements
`mylist.append(c)`	append element to list
`mylist.extend(elems)`	extend list with elems (a list)
`mylist.count(item)`	count how many of an item is in a list
`myliist.index(item)`	at which index is item in a list
`mylist.insert(i, item)`	insert item at position i in a list
`mylist.remove(item)`	remove first occurrence of item in list

```
ls.reverse()
ls.sort()
ls.sort(reverse=True)
```

ZIPPERING 🤐

zip allows you to put two lists together, like a zipper. The result is a list of tuples.

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

The asterisk (*) command, used with zip can unzip:

Try

```
d = zip(*c)
e = zip(*d)
```

DICTIONARIES 

Dictionaries are similar to lists, except that the elements, rather than being indexed by integers, they are indexed by "keys". Keys may be numbers, strings, or objects. The contents of dictionaries can be strings, values, lists, etc.

3 ways for creating a dictionary:

Method 1: curly brackets

```
d = {'quarterback': 'Brady', 'coach': 'Belichik',
'wins':5}
```

keys: quarterback, coach, wins

values: Brady, Belichik, 5

DICTIONARIES 

Method 2: keywords

```
e = dict(first = 'salad', second = 'soup',
         third = 'main course')
```

Method 3: lists

```
f = dict([['country','Canada'],
         ['flag','maple leaf'],['population':3e7]])
```

ADDING ELEMENTS TO A DICTIONARY

```
e = dict(first = 'salad', second = 'soup',
         third = 'main course')
```

We can add elements to an existing dictionary by using the assignment operator, giving a new key and value pair

```
e[fourth = 'dessert']
```

You can also create an empty dictionary, and then add:

```
g = {}
g['fourth'] = 'Taco Bell'
```

FUNCTIONS AND METHODS FOR DICTIONARIES

`len(e)`	number of key/value pairs
`del e['third']`	delete a key/value pair
`k in e`	checks to see whether e contains an item with the key given by k returns True or False
`e.clear()`	clears a dictionary
`e.copy()`	returns the copy of a dictionary
`e.keys()`	returns the keys in a dictionary
`e.items()`	returns a list containing tuples of all of the key/value pairs
`e.values()`	returns the values in the dictionary

WHAT'D WE LEARN? 🤓

objects may have methods or attributes that can be called to get information about a variable or run a "subroutine" on that variable.

There are many ways to collect information in python

- **lists** are flexible data structures to collect information, which you can change later
- **tuples** are data structures to collect information, which you can't change once they're defined (useful for making sure information is not overwritten in your code)
- **dictionaries** are flexible data structures that are defined by key/value pairs