Introductory Python 2

In the previous lecture, we saw *variables, control flow, and functions*. Among variables, we only looked at the basic types: integers, floating point numbers, and strings. Here, we will discuss more complex types:

- Lists
- Tuples
- Dictionaries

Lists

A list is a sequence of items, just like a string is a sequence of characters. The difference is that a list can have items of arbitrary type.

In [1]:
```
# A list of numbers
x = [10, 20, 30]

# A list of strings
y = ['a', 'list', 'of', 'strings']

# A composite list
z = [1, 'composite', 'list']

print('x =', x)
print('y =', y)
print('z =', z)
```

```
x = [10, 20, 30]
y = ['a', 'list', 'of', 'strings']
z = [1, 'composite', 'list']
```

In [2]:
```
# You can even have a list of lists
w = [x, y, z]
print('The combined list is', w)
```

```
# Each item of w is a list, so w only has 3 items
print('The length of w is', len(w))
```

```
The combined list is [[10, 20, 30], ['a', 'list', 'of', 'st
rings'], [1, 'composite', 'list']]
The length of w is 3
```

What use is a list?

- Lists are mainly to organize a set of items on which you plan to do further processing
  - A list of item prices, on which you will do an inflation adjustment
  - A list of share prices for a company, from which you want to figure out daily returns
  - A list of investment accounts on which you want to do detailed portfolio analysis
- Lists are *efficient*
  - List storage is cheap, and processing is fast

We will discuss lists a lot:

- How to access individual items in a list
- How to modify a list
- What are the common operations on lists, and tools for the same
- How do we iterate over all list items
- How do we copy lists

Lists are a critical data structure, so it will be worth our while to spend time on them.

How can we access items in a list

Lists are *indexed* from 0 onwards:

- the first element of a list x is x[0],
- the second element is x[1],
- and so on...

In [3]:
```
print('x =', x)
```

```
x = [10, 20, 30]
```

In [4]:
```
print('The first element is x[0] =', x[0])
print('The length of the list is', len(x))
print('The last element is', x[len(x) - 1])  # NOT x[len(x)]
```

```
The first element is x[0] = 10
The length of the list is 3
The last element is 30
```

There is a simpler way to access the last elements of a list...

In [5]:
```
# A short-form for x[len(x) - 1] is x[-1]
print('The last element is', x[-1])
print('The second-last element is', x[-2])
```

```
The last element is 30
The second-last element is 20
```

Finally, instead of individual items, you can get *slices* of a list.

In [6]:
```
t = ['a', 'b', 'c', 'd', 'e', 'f']
print('t[1:4] =', t[1:4])
```

```
t[1:4] = ['b', 'c', 'd']
```

Here, t[1:4] means: create a new list whose first element is t[1], the second is t[2], all the way up to t[4-1]=t[3].

In [7]:
```
t[1:1] # empty slice
```

Out[7]:
```
[]
```

In [8]:

```
t[0:len(t)] # full list
```

Out[8]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

In [9]:
```
t[1:-1]
```

Out[9]:
```
['b', 'c', 'd', 'e']
```

In [10]:
```
t[0:len(t):2]  # get every second item
```

Out[10]:
```
['a', 'c', 'e']
```

We often need *the first few items* or *the last few items* of a list, so there's a special short-form for these.

In [11]:
```
print(t[:5])   # Same as t[0:5]
print(t[3:])   # Same as t[3:len(t)]
print(t[:])    # Same as t[0:len(t)]
```

```
['a', 'b', 'c', 'd', 'e']
['d', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Last three elements of a list?

In [12]:
```
print(t[-3:])
```

```
['d', 'e', 'f']
```

How do we modify a list?

We can modify any items we can access in a list.

In [13]:
```
print('t =', t)
t[0] = 'A'          # Modify first item
t[-1] = 'F'         # Modify last item
t[2:4] = ['C', 'D'] # Modify a slice
print('modified t =', t)
```

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
modified t = ['A', 'b', 'C', 'D', 'e', 'F']
```

Common list operations

Remember how the + operator on strings *concatenated* strings?

In [14]:
```
print('Mister' + 'Brown')
```

```
MisterBrown
```

It has the same effect on lists.

In [15]:
```
print(['The', 'brown'] + ['fox', 'jumped'] + ['over', 'the', 'fence'])
```

```
['The', 'brown', 'fox', 'jumped', 'over', 'the', 'fence']
```

Similarly, the * operator *repeats* items.

In [16]:
```
print('hey ' * 3)
print([1, 2, 3] * 3)
```

```
hey hey hey 
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In [17]:
```
# sort arranges elements from low to high
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
print(t)
```

```
['a', 'b', 'c', 'd', 'e']
```

Something interesting happened here. We called the sort *method* on the list *object* t. The list is not just a sequence of items; it also comes with a bunch of built-in functions to modify or operate on that list. So the list t is really a combination of

- the list items, that we provide, and
- the functions (also called *methods*)

To emphasize this, we sometimes say that a list is an *object*.

Note that **sort() actually changes the list**. If we don't want to change the list, we can use *sorted()*.

In [18]:
```
t = ['d', 'c', 'e', 'b', 'a']
print('Sorted version of t =', sorted(t))
print('t remains untouched: t =', t)
```

```
Sorted version of t = ['a', 'b', 'c', 'd', 'e']
t remains untouched: t = ['d', 'c', 'e', 'b', 'a']
```

Both *sort()* and *sorted()* default to sorting in ascending order, but they have an option to sort in descending order.

In [19]:
```
print('t =', t)
print('Descending via sorted =', sorted(t, reverse=True))
```

```
t = ['d', 'c', 'e', 'b', 'a']
Descending via sorted = ['e', 'd', 'c', 'b', 'a']
```

In [20]:
```
t.sort(reverse=True)
print(t)
```

```
['e', 'd', 'c', 'b', 'a']
```

But how do we sort if the list has more complicated items? Say, a list of lists...

In [21]:
```
t = []
t.append([100000, 3.4])   # mortgage, and mortgage rate (percentage)
t.append([30000, 2.9])
t.append([80000, 2.1])
t.append([110000, 2.8])
print(t)
```

```
[[100000, 3.4], [30000, 2.9], [80000, 2.1], [110000, 2.8]]
```

Suppose we want to sort by mortgage rate (ignoring the mortgage amount). How do we do this?

 sorted

All this gets done by just writing: `sorted(t, key=getMortgageRate)`

So we just need to write a **getMortgageRate** function that

- can take any item of the list t as input
  - i.e., something like [100000, 3.4]
- and returns the value on which we want to sort
  - from [100000, 3.4], return 3.4

In [22]:
```
def getMortgageRate(list_item):
    """Given a list_item of type [mortgage_amount, mortgage_rate],
       return the mortgage_rate."""
    return list_item[1]
```

In [23]:
```
sorted(t, key=getMortgageRate)
```

Out[23]:
```
[[80000, 2.1], [110000, 2.8], [30000, 2.9], [100000, 3.4]]
```

Example: Sort by descending order of yearly mortgage interest.

In [24]:
```
def yearly_interest(list_item):
    """list_item is [mortgage_amount, mortgage_rate]
       Yearly payment = amount * rate / 100.0"""
    return list_item[0] * (list_item[1] / 100.0)
```

In [25]:
```
sorted_list = sorted(t, key=yearly_interest, reverse=True)
for i in sorted_list:
    print('amount =', i[0], ' rate =', i[1], 'yearly interest =', i[0]
```

```
amount = 100000  rate = 3.4 yearly interest = 3400.0000000000005
amount = 110000  rate = 2.8 yearly interest = 3079.9999999999995
amount = 80000  rate = 2.1 yearly interest = 1680.0
amount = 30000  rate = 2.9 yearly interest = 869.9999999999999
```

Appending and deleting items

In [26]:
```
# Some common methods
t = []  # empty list

# Appending elements
t.append('a')
t.append('b')
t.append('c')
print('After appending:', t)

# Deleting elements
item = t.pop()
print('Deleted item', item, 'from end of the list')
print('New list =', t)
```

```
After appending: ['a', 'b', 'c']
Deleted item c from end of the list
New list = ['a', 'b']
```

There are several such methods, not just for lists but also for strings and other data types. Just call help() on the object you wish to study.

In [27]:
```
help(t)
```

```
Help on list object:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new
empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
```

```
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate
signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
```

```
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the list.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(self, /)
 |      Return the size of the list in memory, in bytes.
 |
 |  append(self, object, /)
 |      Append object to the end of the list.
 |
 |  clear(self, /)
 |      Remove all items from list.
 |
 |  copy(self, /)
 |      Return a shallow copy of the list.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  extend(self, iterable, /)
 |      Extend list by appending elements from the iterabl
e.
 |
 |  index(self, value, start=0, stop=9223372036854775807,
/)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  insert(self, index, object, /)
 |      Insert object before index.
 |
```

```
 |  pop(self, index=-1, /)
 |      Remove and return item at index (default last).
 |
 |      Raises IndexError if list is empty or index is out
of range.
 |
 |  remove(self, value, /)
 |      Remove first occurrence of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  reverse(self, /)
 |      Reverse *IN PLACE*.
 |
 |  sort(self, /, *, key=None, reverse=False)
 |      Stable sort *IN PLACE*.
 |
 |  ----------------------------------------------------
---------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for
accurate signature.
 |
 |  ----------------------------------------------------
---------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

Iterating on a list

A common idiom is to process every item in a list. We've already seen how the *for-loop* does this.

In [28]:
```
t = ['a', 'B', 'c', 'D']
for item in t:
    print('item =', item, 'capitalized item =', item.upper())
```

```
item = a capitalized item = A
```

```
item = B capitalized item = B
item = c capitalized item = C
item = D capitalized item = D
```

In [29]:
```
# enumerate() gives the list index and list item together
for index, item in enumerate(t):
    print('Item number ', index, 'is ', item)
```

```
Item number  0 is  a
Item number  1 is  B
Item number  2 is  c
Item number  3 is  D
```

In [30]:
```
# The "in" operator checks if an item belongs to the list
print("Does 'a' belong to the list?", ('a' in t)) # Note use of parenth
print("Does 'z' belong to the list?", ('z' in t))
```

```
Does 'a' belong to the list? True
Does 'z' belong to the list? False
```

Copying lists

Making a copy of a list has traps for the unwary!

In [31]:
```
a = [1, 2, 3, 4]
b = a       # Create a "copy"
a[1] = 100  # Modify one list
print(a)
print(b)
```

```
[1, 100, 3, 4]
[1, 100, 3, 4]
```

Both lists were modified! This is because, internally, 'a' is a pointer to a memory location where the list is stored, and 'b=a' just sets b to point to the same location. So modifying a also modifies b.

If you want to create a completely independent copy of a, use the following:

In [32]:
```
b = list(a)
```

```
a[1] = 500  # This updates a, but not b
print('a =', a)
print('b =', b)
```

```
a = [1, 500, 3, 4]
b = [1, 100, 3, 4]
```

It gets more difficult when the object you are copying is more complicated, so we can also use a special tool just to make such completely independent copies (called *deep* copies).

In [33]:

```
# import a special module, called the "copy" module
import copy

# create the independent copy using the deepcopy() method
c = copy.deepcopy(a)

print('a=', a, 'and c=', c)

# change a
a[2] = 10

# check that this did not change c
print('After updating a: a=', a, 'and c=', c)
```

```
a= [1, 500, 3, 4] and c= [1, 500, 3, 4]
After updating a: a= [1, 500, 10, 4] and c= [1, 500, 3, 4]
```

Summary of Lists

- **Creation:** We can create lists using
  - *square brackets:* x = [1,2,3]
  - *list:* x = list(some_other_list)

- **Access:** We can access and modify list items
  - x[0:2] = ['a', 'b']

- **Common operations:**
  - inclusion via *in*
  - iteration via *for*

- *sort*
- *append* and *pop*

Careful with *sort*! It changes the list!

Percentage changes in list

You are given a list of stock prices over days. Compute the return for each day.

$$\text{today's return} = \frac{\text{today's price} - \text{yesterday's price}}{\text{yesterday's price}}$$

Assume the first day's return is 0.0

In [34]:
```
def returns(prices):
    """Daily returns on prices"""
    daily_returns = [0.0]
    for i in range(len(prices) - 1):
        return_for_day_i_plus_1 = (prices[i + 1] - prices[i]) / prices[
        daily_returns.append(return_for_day_i_plus_1)
    return daily_returns

# Test
returns([100, 101, 102, 103, 102, 101, 100])
```

Out[34]:
```
[0.0,
 0.01,
 0.009900990099009901,
 0.00980392156862745,
 -0.009708737864077669,
 -0.00980392156862745,
 -0.009900990099009901]
```

Tuples

Tuples are just like lists, except the values of their items are fixed.

In [35]:
```
# Create a tuple
x = ('John', 'Doe', 123)
```

```
print(x)
```

```
('John', 'Doe', 123)
```

In [36]:

```
# Tuple assignment
firstname, lastname, employee_id = x
print('Welcome ', firstname, lastname, '!')
```

```
Welcome  John Doe !
```

In [37]:

```
def employee_data(t):
    """Gets a tuple of the form (firstname, lastname, ID).
       Prints these out separately."""
    print('Employee name is', t[0], t[1])
    print('ID =', t[2])

x = ('John', 'Doe', 123)
employee_data(x)

employee_data(('Alice', 'Bach', 35))
# Note, we need two sets of parentheses, one for the function employee_
# and one for the tuple ('Alice', 'Bach', 35)
```

```
Employee name is John Doe
ID = 123
Employee name is Alice Bach
ID = 35
```

Remember that tuples cannot be modified.

```
x = ('John', 'Doe', '123')
x[0] = 'Johnny'  # is wrong!
```

Why ever use tuples and not just lists?

- Can't lists hold the same items that tuples can?
  - Yes
- Can't lists be updated whenever we want?
  - Yes

But tuples might be better for *error checking*

- Maybe I *know* the number of items beforehand
  - I will only store (latitude, longitude)
  - My tuples will only have a person's (name, age, gender)
- Maybe these values are fixed forever
  - I want my code to *ensure* that they never change

Dictionaries

Suppose I want to convert English to Spanish. How do I do it?

- I look up a dictionary that contains, for each English word, the equivalent Spanish word.

More generally, a dictionary is a map from *keys* to *values*

- *key* = an English word, *value* = corresponding Spanish word
- *key* = bank customer's ID, *value* = the customer's balance
- *key* = a date, *value* = Google's stock price on that date

In [38]:
```
eng2sp = {} # empty dictionary
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
print(eng2sp)

# We could also have initialized this as follows
eng2sp = {'one':'uno', 'two':'dos'}
```

```
{'one': 'uno', 'two': 'dos'}
```

There is *one* difference between a Python dictionary and a real English-to-Spanish dictionary.

- In the real world, the keys of a dictionary are ordered (the dictionary goes from Aardvark to Zymosan).
- In Python dictionaries, the order of the keys is garbled

- the order in which you add items to a dictionary does not matter
- there is no "first item" or "last item" of a dictionary,
- unlike for a list or tuple.

Accessing items in a dictionary

If the key is known, getting the dictionary value is trivial

In [39]:
```
def english_to_spanish(eng2sp, word):
    """Convert an English word to Spanish"""
    if word in eng2sp:
        print('The Spanish for', word, 'is', eng2sp[word])
    else:
        print(word, 'is not a recognized English word')

english_to_spanish(eng2sp, 'two')
english_to_spanish(eng2sp, 'dos')  # will not work; 'dos' is a value, r
```

```
The Spanish for two is dos
dos is not a recognized English word
```

We used the "in" operator (word in eng2sp) to check if word is a valid key of the eng2sp dictionary. Once we knew that it is a valid key, we used eng2sp[word] to access its value.

NOTE: It is easy to go from key to value, but hard to go from value to the corresponding key(s). Think about the real English-to-Spanish dictionary.

We can get lists of

- all the keys via the *keys()* method
- all the values via the *values()* method
- all (key, value) pairs via the *items()* method

In [40]:
```
print(list(eng2sp.keys()))  # Returns a list of keys
```

```
['one', 'two']
```

In [41]:
```
print(list(eng2sp.values())) # Returns the list of values
```

```
['uno', 'dos']
```

In [42]:
```
print(list(eng2sp.items())) # Returns all (key, value) pairs as a list
```

```
[('one', 'uno'), ('two', 'dos')]
```

Iterating over a dictionary

Say we want to create a new dictionary that capitalizes the Spanish words. There are several ways to do it.

In [43]:
```
# FIRST WAY: Iterate over the list of keys
eng2sp_cap = {}
for key in eng2sp.keys():
    value = eng2sp[key]
    capitalized_value = value.upper()  # value is a string, and we use
    eng2sp_cap[key] = capitalized_value
print('Capitalized dictionary:', eng2sp_cap)
```

```
Capitalized dictionary: {'one': 'UNO', 'two': 'DOS'}
```

What we did was:

1. we iterated over the keys
2. for each key, we found the corresponding value
3. we processed this key-value pair.

This is so common that there is a short-form for doing this.

In [44]:
```
# SECOND WAY: Iterate over key-value pairs
eng2sp_cap = {}
for key, value in eng2sp.items(): # this iterates over (key, value) tup
    capitalized_value = value.upper()
    eng2sp_cap[key] = capitalized_value
print('Capitalized dictionary:', eng2sp_cap)
```

```
Capitalized dictionary: {'one': 'UNO', 'two': 'DOS'}
```

The difference between the two methods is in eng2sp.keys() versus eng2sp.items().

- eng2sp.keys() gives a *list* of the keys
- eng2sp.items() gives a *list* of (key, value) tuples

When we say

```
for key, value in eng2sp.items():
```

we are assigning the variables key and value using this tuple.

Summary of dictionaries versus lists

Operation	Dictionary	List
access item	dict[key]	list[index]
update item	dict[key] = value	list[index] = value
check if item exists	if key in dict	if item in list
slices	*not possible*	list[3:5]

You use dictionaries when

- you will need to access individual items
  - and not just iterate over all items
- there is some key for each item, and the value can be arbitrary

You use lists when

- you have a particular ordering of items in mind, or
- you just want to iterate over a bunch of items

List comprehensions

A common coding paradigm is the following:

```
new_list = []
for item in old_list:
    new_list.append( some_function(item) )
```

There is a short-form for this sequence:

```
new_list = [ some_function(item) for item in old_list ]
```

Given a list, create a new list of the squares of the items

In [45]:
```
orig_list = [1, 2, 3, -1, -2]
new_list = [item * item for item in orig_list]
print(orig_list)
print(new_list)
```

```
[1, 2, 3, -1, -2]
[1, 4, 9, 1, 4]
```

Another variant involves an if-statement:

```
new_list = []
for item in old_list:
    if some_condition(item):
        new_list.append( some_function(item) )
```

The short form is

```
new_list = [ some_function(item) for item in old_list if
some_condition(item) ]
```

Create a list of square-roots of the positive elements in a given list

In [46]:
```
import math
new_list = [(item, math.sqrt(item)) for item in orig_list if item > 0]
print(new_list)
```

```
[(1, 1.0), (2, 1.4142135623730951), (3, 1.732050807568877
2)]
```

Find the smallest number in a list that is bigger than a query number

If the list is [5, 3, 4, 9, 1] and the query is 2.1, return 3.

If there's no such item, return -1.

Here's the easy way:

- Sort the list in ascending order
- Iterate over it
  - until you hit the first item greater than the query

In [47]:
```
def smallest_larger_than(number_list, query):
    """Return the smallest number in number_list that is
    greater than the query number."""

    for item in sorted(number_list):
        if item > query:
            return item

    # We went through the list, and nothing was greater than query
    return -1

# Test
print(smallest_larger_than([5,4,3,9,1], 2.1))   # Should give 3
print(smallest_larger_than([], 2.1))            # Should give -1
```

```
3
-1
```

However, sorting is more than we needed! How do we do it without sorting?

We will have a tracker variable that keeps the smallest number seen so far greater than the query.

- Iterate over list
- Update the tracker whenever some item is greater than query AND

- either tracker hasn't been initialized yet, OR
- the new item is smaller than the tracker value (best answer seen so far)

In [48]:
```
def smallest_try_2(number_list, query):
    # tracker hasn't been initialized to a valid value until we
    # see some list item greater than query
    tracker_initialized = False
    tracker = 0

    # Iterate over the list
    for i in number_list:
        if i > query:
            if not(tracker_initialized) or (i < tracker):
                tracker = i
                tracker_initialized = True

    if tracker_initialized:
        return tracker
    else:
        return -1

# Test
print(smallest_try_2([5,4,3,9,1], 2.1))   # Should give 3
print(smallest_try_2([], 2.1))            # Should give -1
```

```
3
-1
```