Python Programming for Data Science

Week 37, Monday

More container types

- Tuples
- Dictionaries

Which container types to use when

String formatting

Recap: datatypes — a popquiz

- How do you find the length of a string?
- How do you extract a substring from a string?
- What is the difference between a function and a method?
- What does the `strip` string-method do?
- What is the difference between a for-loop and a while loop?
- What is the difference between `sort()` and `sorted()`
- What does the range function do?

Tuples

Types: Tuples

Tuples are immutable lists. This means that you cannot alter them in any way after you have created them (just like strings).
Tuples are defined using commas, and often using parentheses:

```
t = (1,2,3,4)
t = 1,2,3,4     # parenthesis can be ommitted
```

Example:

```
x = 3
y = 4
print((x,y))   # Making a tuple on-the-fly
```

```
output

(3, 4)
```

Types: Tuples — assigning

You can assign to multiple variables at once by using a tuple on the left-hand side of the assignment

```
my_tuple = (1,2)     # Creating a tuple
x, y = my_tuple      # Assigning variable x to 1 and
variable y to 2
```

Another example:

```
x,y = y,x        # Nice trick: swapping
the values
print((x,y))     # of x and y
```

output

```
(2, 1)
```

Types: Tuples — operators

Tuples support a subset of the operators and methods of lists (only ones that don't modify the collection):

```
+, *, [], ==, !=, <, <=, >, >=, in
```

Just as with strings, you cannot assign to an entry in a tuple (due to immutability):

```
t = (1,2,3,4)
t[2] = 1
```

```
output

Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support item
assignment
```

Types: Tuples — why?

Why have tuples when we already have lists?

- Tuples are smaller and faster.
- Tuples are used as return values from functions.
- The fact that tuples are immutable makes it possible to use them as keys in a dictionary. More about this later today.

Tuples are a streamlined version of lists. If you know in advance that your sequences are fixed (e.g. coordinates in space) then use tuples.

Tuples — exercise

1. Create a tuple called `triplet` of size 3 containing the numbers 1, 2, and 3.
2. Create three variables `x`, `y` and `z` and use the tuple `triplet` to initialize them.
3. Now create a tuple called `quartet` of size 4 (with the numbers 1,2,3,4), and try it again. What happens?
4. Tuples support slicing. Can you use this to fix the problem?

Tuples — exercise — solution

1. Create a tuple called `triplet` of size 3 containing the numbers 1, 2, and 3.

```
triplet = (1,2,3)
```

2. Create three variables x, y and z and use the tuple `triplet` to initialize them.

```
x,y,z = triplet
```

3. Now create a tuple called `quartet` of size 4, and try it again. What happens?

```
quartet =
(1,2,3,4)
x,y,z =
quartet
```

```
output

Traceback (most recent call last):
  File "/home/lpp/lpp2016/test.py",
line 2, in <module>
    x,y,z = quartet
ValueError: too many values to
unpack
```

Tuples — exercise — solution (2)

1. Tuples support slicing. Can you use this to fix the problem?

```
x,y,z = quartet[:3]    # Only use index 0,1,2
```

or

```
x,y,z,_ = quartet      # _ is often used as a dummy variable
```

Dictionaries

Types: Dictionaries

Recall: lists give you access to a value through their index:

```
my_list = ["a", "b", "c"]
print(my_list[1])    # lookup using index. Index 1 => "b"
```

Dictionaries are similar, but more general: they give access to values by their *key*, which can for instance be strings. Dictionaries consist of (key, value) pairs. They are defined using curly brackets with ":" as key/value separator: Example:

```
days = {'Mon':'Monday', 'Tue':'Tuesday',
'Wed':'Wednesday'}
```

Idea: associate weekdays with their abbreviations

Types:Dictionaries — accessing elements

You can look up an element in a dictionary using []:

```
days = {'Mon':'Monday', 'Tue':'Tuesday',
'Wed':'Wednesday'}
print(days['Mon'])
```

```
output
'Monday'
```

Note the similarity to lists, except that individual elements are not associated with an index number, but a *key*.
You will get a KeyError if you use a key that doesn't exist:

```
print(days['Fri'])
```

```
output
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(days['Fri'])
KeyError: 'Fri'
```

Types: Dictionaries — inserting and deleting elements

You can add new elements to a dictionary using the [] operator:

```
days = {}   # empty dictionary
days['Mon'] = 'Monday'
print(days)
```

```
output

{'Mon': 'Monday'}
```

Elements can be deleted using the `del` statement:

```
del days['Mon']
print(days)
```

```
output

{}
```

Types: Dictionaries are not ordered

One important difference from lists, is that dictionaries are not ordered

```
days = {'Mon':'Monday', 'Tue':'Tuesday',
'Wed':'Wednesday'}
print(days)
```

```
{'Tue': 'Tuesday', 'Mon': 'Monday', 'Wed': 'Wednesday'}
```

Take home message: You cannot rely on the dictionary maintaining the order in which you insert elements - you can only access them by their key.
This has changed in recent versions of Python (3.7+). They are now ordered by order of insertion.

Types: Dictionaries — operators

The only really meaningful operators for dictionaries are

```
==, !=, [], in
```

which provide the same functionality as they do in lists

Types: Dictionaries — methods

Some interesting dictionary methods:

`get`	Similar to `[]` - without key errors
`items`	Return list of (key,value) tuples
`keys`	Return list of keys
`pop`	Remove specified key and return value
`popitem`	Remove specified key and return (key,value) tuple
`setdefault`	Same as get, but add (key, None) if key is not already there
`update`	Add all elements from specified dictionary to current dictionary
`values`	Return list of values

```
>>> days.keys()
['Tue', 'Mon', 'Wed']
>>> days.pop('Wed')
'Wednesday'
>>> 'Wed' in days
False
```

Dictionaries — exercise

1. Create a dictionary with the keys "firstname", "lastname", and "age", and appropriate values.
2. Add a key named "address" to this dictionary.
3. Print out the list of keys in your dictionary.
4. Create a "name" key which as value contains a string with both your first and last names. Then remove the first name and last name keys. Can you do this without actually typing in your name again?

Dictionaries — exercise — solution

1. Create a dictionary with the keys firstname, lastname, and age, and appropriate values.

```
personal_info = {"firstname": "Donald",
                 "lastname": "Trump",
                 "age": 74}
```

2. Add an `address` key to this dictionary.

```
personal_info["address"] = "Whitehouse"
```

3. Print out the list of keys in your dictionary.

```
print(personal_info.keys())
```

```
output
dict_keys(['firstname', 'lastname', 'age', 'address'])
```

Dictionaries — exercise — solution (2)

4. Create a "name" key which as associated value contains a string with both your first and last names. Then remove the first name and last name keys. Can you do this without actually typing your name again?

```
personal_info["name"] = personal_info["firstname"] + "
" + \
                        personal_info["lastname"]
del personal_info["firstname"]
del personal_info["lastname"]
print(personal_info)
```

```
output
{'age': 74, 'name': 'Donald Trump', 'address':
'Whitehouse'}
```

Or:

```
personal_info["name"] = personal_info.pop("firstname")
+ " " + \
                        personal_info.pop("lastname")
```

The NoneType

Types - the None value

It is possible to specify an empty value – the value `None`. `None` is often returned from functions as an empty result:

```
print(days.get('bla'))  # Get returns None if the key is not found
```

```
output
None
```

Collection types: which should I use?

Collection types – an overview

We covered various collection types last week, but when do you use what?

- lists
- tuples
- dictionaries

It can be very important what collection type you choose for your data.

Here are some questions you could ask yourself:

Collection types – lists vs. dictionaries

Is my data unordered or ordered?

Examples:

- The lines in a file are often naturally ordered, and most efficiently stored in a list.
- Experimental results on different genes might be unordered, and could be stored in a dictionary using gene-names as keys.

Note: Sometimes it makes sense to use a dictionary to store ordered data - when the data is sparse.

Collection types – lists vs. tuples

Is the collection size naturally fixed?

Tuples should be used when the type of data you are working with always has the same number of items, and the items don't need to be changed individually

Examples:

- 3D-coordinates
- Each line of a file containing experimental results with a fixed number of columns

String formatting

Types - Strings - formatting

You can insert values at certain places in a string using *string formatting*.
Two ways of string formatting in Python:

- old way - operator %
- new way - str.format()

There is an *even newer* way, using so-called f-strings. This technique is still less common than the others, so we'll not cover it here.

String formatting - % operator

```
s = "Name: %s, Height: %f" % ("Barack Obama", 1.85)
print(s)
```

```
output
'Name: Barack Obama, Height: 1.85'
```

%s and %f are called *conversion specifiers*, and tell Python how the values should be interpretted

Most commonly used conversion specifiers:

%s	string
%d	integer
%f	float

String formatting - % operator (2)

In most cases, you can just use the conversion specifier %s, which will use the default string representation of the value that you are inserting

```
print("Name: %s, Height: %s" % ("Barack Obama", 1.85))
```

```
output
'Name: Barack Obama, Height: 1.85'
```

...but for detailed control of the output, use the appropriate conversion specifier:

```
# print height with 4 digits after comma
print("Name: %s, Height: %0.4f" % ("Barack Obama", 1.85))
```

```
output
'Name: Barack Obama, Height: 1.8500'
```

More on conversion specifiers: the **Python documentation**.

String formatting - % operator - by name

You can also refer to the values by name

```
print("I have %(n_apples)s apples and %(n_pears)s pears."
% {"n_apples":5,
"n_pears":7})
```

```
output
'I have 5 apples and 7 pears.'
```

Note the use of a dictionary on the right.
This can be convenient if you want to insert the same value many times:

```
print("3%(sep)s14 2%(sep)s72 1%(sep)s41" % {"sep":"."})
```

```
output
'3.14 2.72 1.41'
```

String formatting - .format()

Python 3 introduced a new string formatting technique:

```
"Name: {}, Height: {}".format("Barack Obama", 1.85)
```

```
output

'Name: Barack Obama, Height: 1.85'
```

Within the curly brackets, you can:

Specify a positional index:

```
"Name: {1}, Height: {0}".format("Barack Obama", 1.85)
```

```
output

'Name: 1.85, Height: Barack Obama'
```

Conversion to specific type

```
"Name: {:s}, Height: {:f}".format("Barack Obama", 1.85)
```

```
output

'Name: Barack Obama, Height: 1.850000'
```

Refer by name

```
"Name: {name}, Height: {height}".format(name="Barack Obama", height=1.85)
```

```
output

'Name: Barack Obama, Height: 1.85'
```

...and any combination of these (and much more)

Types - Strings - formatting - Exercise

1. Create a string with your firstname, lastname and age, just like last week, but now using string formatting, using first the %-style and then the `.format` style. The output should look like this:

```
Name: myfirstname mylastname
Age: age
```

2. The `range` function creates a list of consecutive numbers. E.g.:

```
print(list(range(4)))
```

```
output
[0, 1, 2, 3]
```

Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

Types - Strings - formatting - Exercise - solution

1. Create a string with your firstname, lastname and age, just like before, but now using string formatting — and referring to the values *by name*. The output should look like this:

```
output
Name: Barack Obama
                Age: 60
```

Answer:

```
print("Name: %s %s \nAge: %d" % ("Barack", "Obama",
60))
```

or

```
print("Name: {} {} \nAge: {:d}".format("Barack",
"Obama", 60))
```

Types - Strings - formatting - Exercise - solution (2)

2. Try to explain what the following code does:

```
print(("%s\n"*5)%tuple(range(5)))
```

```
print("%s\n"*5)

print(tuple(range(5)))

print('%s\n%s\n%s\n%s\n%s\n
' %
    (0, 1, 2, 3, 4))
```

output

```
# Repeat string 5 times
'%s\n%s\n%s\n%s\n%s\n'

# Create [0,1,2,3,4] list
and
# convert it to a tuple
(0, 1, 2, 3, 4)

# print numbers on separate
lines
0
1
2
3
4
```

Speaker notes