(A very brief) Introduction to Python

Lecture 02

Dr. Colin Rundel

Basic types

Type system basics

Like R, Python is a dynamically typed language but the implementation details are very different as it makes extensive use of an object oriented class system for implementation (more on this later)

```
1 True
```

```
True
```

```
1 type(True)
```

```
<class 'bool'>
```

```
1 1
```

```
1
```

```
1 type(1)
```

```
<class 'int'>
```

```
1 1.0
```

```
1.0
```

```
1 type(1.0)
```

```
<class 'float'>
```

```
1 1+1j
```

```
(1+1j)
```

```
1 type(1+1j)
```

```
<class 'complex'>
```

```
1 "string"
```

```
'string'
```

```
1 type("string")
```

```
<class 'str'>
```

Note - all of these types are *scalar* values.

Dynamic types

As just mentioned, Python is dynamically typed langiage so most basic operations will attempt to coerce object to a consistent type appropriate for the operation.

Boolean operations:

```
1 and True
```

```
True
```

```
0 or 1
```

```
1
```

```
not 0
```

```
True
```

```
not (0+0j)
```

```
True
```

```
not (0+1j)
```

```
False
```

Comparisons:

```
5. > 1
```

```
True
```

```
5. == 5
```

```
True
```

```
1 > True
```

```
False
```

```
(1+0j) == 1
```

```
True
```

```
"abc" < "ABC"
```

```
False
```

Mathematical operations

```
1 1 + 5
```

```
6
```

```
1 1 + 5.
```

```
6.0
```

```
1 1 * 5.
```

```
5.0
```

```
1 True * 5
```

```
5
```

```
1 (1+0j) - (1+1j)
```

```
-1j
```

```
1 5 / 1.
```

```
5.0
```

```
1 5 / 2
```

```
2.5
```

```
1 5 // 2
```

```
2
```

```
1 5 % 2
```

```
1
```

```
1 7 ** 2
```

```
49
```

Coercion errors

Python is not quite as liberal as R when it comes to type coercion,

```
1 "abc" > 5
```

```
TypeError: '>' not supported between instances of 'str' and 'int'
```

```
1 "abc" + 5
```

```
TypeError: can only concatenate str (not "int") to str
```

```
1 "abc" + str(5)
```

```
'abc5'
```

```
1 "abc" ** 2
```

```
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

```
1 "abc" * 3
```

```
'abcabcabc'
```

More on why this happens in a little bit...

Casting

Explicit casting between types can be achieved using the types as functions, e.g. `int()`, `float()`, `bool()`, or `str()`.

```
1 float("0.5")
```

```
0.5
```

```
1 float(True)
```

```
1.0
```

```
1 int(1.1)
```

```
1
```

```
1 int("2")
```

```
2
```

```
1 int("2.1")
```

```
ValueError: invalid literal for int() with base 10:
'2.1'
```

```
1 bool(0)
```

```
False
```

```
1 bool("hello")
```

```
True
```

```
1 str(3.14159)
```

```
'3.14159'
```

```
1 str(True)
```

```
'True'
```

Variable assignment

When using Python it is important to think of variable assignment as the process of attaching a name to an object (literal, data structure, etc.)

```
1 x = 100
2 x
```

```
100
```

```
1 x = "hello"
2 x
```

```
'hello'
```

```
1 ß = 1 + 2 / 3
2 ß
```

```
1.6666666666666665
```

```
1 a = b = 5
```

```
1 a
```

```
5
```

```
1 b
```

```
5
```

Python variable names can be of any length, and must only contain letters, numbers and underscores. They may not begin with a number nor conflict with language keywords. Python 3 supports a subset of unicode for variable

string literals

Strings can be defined using a couple of different approaches,

```
1 'allows embedded "double" quotes'
```

```
'allows embedded "double" quotes'
```

```
1 "allows embedded 'single' quotes"
```

```
"allows embedded 'single' quotes"
```

strings can also be triple quoted, using single or double quotes, which allows the string to span multiple lines.

```
1 """line one
2 line two
3 line three"""
```

```
'line one\nline two\nline three'
```

a `\` can also be used to span a long string over multiple lines without including the newline

```
1 "line one \
2 not line two \
3 not line three"
```

f strings

As of Python 3.6 you can use f strings for string interpolation formatting (as opposed to `%`-formatting and the `format()` method).

```
1 x = [0,1,2,3,4]
2 f"{x[::2]}"
```

```
'[0, 2, 4]'
```

```
1 f'{x[0]}, {x[1]}, ...'
```

```
'0, 1, ...'
```

```
1 f"From {min(x)} to {max(x)}"
```

```
'From 0 to 4'
```

```
1 f"{x} has {len(x)} elements"
```

```
'[0, 1, 2, 3, 4] has 5 elements'
```

See PEP 498

raw strings

One other special type of string literal you will come across are raw strings (prefixed with `r`) - these are like regular strings except that `\` is treated as a literal character rather than an escape character.

```
1 print("ab\\cd")
```

```
ab\cd
```

```
1 print("ab\ncd")
```

```
ab
cd
```

```
1 print("ab\tcd")
```

```
ab  cd
```

```
1 print(r"ab\\cd")
```

```
ab\\cd
```

```
1 print(r"ab\ncd")
```

```
ab\ncd
```

```
1 print(r"ab\tcd")
```

```
ab\tcd
```

Special values

By default Python does not support missing values and non-finite floating point values are available but somewhat awkward to use. There is also a `None` type which is similar in spirit and functionality to `NULL` in R.

```
1/0
```

```
ZeroDivisionError: division by zero
```

```
1./0
```

```
ZeroDivisionError: float division by zero
```

```
float("nan")
```

```
nan
```

```
float("-inf")
```

```
-inf
```

```
5 > float("inf")
```

```
False
```

```
5 > float("-inf")
```

```
True
```

```
None
type(None)
```

```
<class 'NoneType'>
```

We will not be using these values much currently, but they will be relevant when discussing pandas in a couple of

Sequence types

lists

Python lists are a *heterogenous*, *ordered*, *mutable* containers of objects (they behave very similarly to lists in R).

```
1 [0,1,1,0]
```

```
[0, 1, 1, 0]
```

```
1 [0, True, "abc"]
```

```
[0, True, 'abc']
```

```
1 [0, [1,2], [3,[4]]]
```

```
[0, [1, 2], [3, [4]]]
```

```
1 x = [0,1,1,0]
2 type(x)
```

```
<class 'list'>
```

```
1 y = [0, True, "abc"]
2 type(y)
```

```
<class 'list'>
```

More on the underlying data structure and its properties next time

Common operations

```
1 x = [0,1,1,0]
2
3 2 in x
```

False

```
1 2 not in x
```

True

```
1 x + [3,4,5]
```

[0, 1, 1, 0, 3, 4, 5]

```
1 x * 2
```

[0, 1, 1, 0, 0, 1, 1, 0]

```
1 len(x)
```

4

```
1 max(x)
```

1

```
1 x.count(1)
```

2

```
1 x.count("1")
```

0

See here and here for a more complete listing of functions and methods.

list subsetting

Elements of a list can be accessed using the `[]` method, element position is indicated using 0-based indexing, and ranges of values can be specified using slices (`start:stop:step`).

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

```
1 x[0]
```

```
1
```

```
1 x[3]
```

```
4
```

```
1 x[0:3]
```

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

```
1 x[3:]
```

```
[4, 5, 6, 7, 8, 9]
```

```
1 x[-3:]
```

```
[7, 8, 9]
```

```
1 x[:3]
```

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

When using a slice the default values are `0:length(x):1`

slice w/ step

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

```
1 x[0:5:2]
```

```
[1, 3, 5]
```

```
1 x[0:6:3]
```

```
[1, 4]
```

```
1 x[0:len(x):2]
```

```
[1, 3, 5, 7, 9]
```

```
1 x[0::2]
```

```
[1, 3, 5, 7, 9]
```

```
1 x[::2]
```

```
[1, 3, 5, 7, 9]
```

```
1 x[::-1]
```

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

Exercise 1

Come up with a slice that will subset the following list to obtain the elements requested:

```
1 d = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Select only the odd values in this list
- Select every 3rd value starting from the 2nd element.
- Select every other value, in reverse order, starting from the 9th element.
- Select the 3rd element, the 5th element, and the 10th element

mutability

Since lists are mutable the stored values can be changed,

```
1 x = [1,2,3,4,5]
```

```
1 x[0] = -1
2 x
```

```
[-1, 2, 3, 4, 5]
```

```
1 del x[0]
2 x
```

```
[2, 3, 4, 5]
```

```
1 x.append(7)
2 x
```

```
[2, 3, 4, 5, 7]
```

```
1 x.insert(3, -5)
2 x
```

```
[2, 3, 4, -5, 5, 7]
```

```
1 x.pop()
```

```
7
```

```
1 x
```

```
[2, 3, 4, -5, 5]
```

```
1 x.clear()
2 x
```

```
[]
```

lists, assignment, and mutability

When assigning an object a name (`x = ...`) you do not necessarily end up with an entirely new object, see the example below where both `x` and `y` are names that are attached to the same underlying object in memory.

```
1 x = [0,1,1,0]
2 y = x
3
4 x.append(2)
```

What are the values of `x` and `y` now?

```
1 x
```

```
[0, 1, 1, 0, 2]
```

```
1 y
```

```
[0, 1, 1, 0, 2]
```

lists, assignment, and mutability

To avoid this we need to make an explicit copy of the object pointed to by `x` and point to it with the name `y`.

```
1 x = [0,1,1,0]
2 y = x.copy()
3
4 x.append(2)
```

What are the values of `x` and `y` now?

```
1 x
```

```
[0, 1, 1, 0, 2]
```

```
1 y
```

```
[0, 1, 1, 0]
```

More on `.copy()` and `.deepcopy()` methods later on in the course.

Value unpacking

lists (and other sequence types) can be unpacking into multiple variables when doing assignment,

```
1 x, y = [1,2]
2 x
```

```
1
```

```
1 y
```

```
2
```

```
1 x, y = [1, [2, 3]]
2 x
```

```
1
```

```
1 y
```

```
[2, 3]
```

```
1 x, y = [[0,1], [2, 3]]
2 x
```

```
[0, 1]
```

```
1 y
```

```
[2, 3]
```

```
1 (x1,y1), (x2,y2) = [[0,1], [2, 3]]
2 x1
```

```
0
```

```
1 y1
```

```
1
```

```
1 x2
```

```
2
```

```
1 y2
```

```
3
```

Extended unpacking

It is also possible to use extended unpacking via the * operator (in Python 3)

```
1 x, *y = [1,2,3]
2 x
```

```
1
```

```
1 y
```

```
[2, 3]
```

```
1 *x, y = [1,2,3]
2 x
```

```
[1, 2]
```

```
1 y
```

```
3
```

If * is not used here, we get an error:

```
1 x, y = [1,2,3]
```

```
ValueError: too many values to unpack (expected 2)
```

tuples

Python tuples are a *heterogenous*, *ordered*, *immutable* containers of values.

They are nearly identical to lists except that their values cannot be changed - you will most often encounter them as a tool for packaging multiple objects when returning from a function.

```
1 (1, 2, 3)
```

```
(1, 2, 3)
```

```
1 (1, True, "abc")
```

```
(1, True, 'abc')
```

```
1 (1, (2,3))
```

```
(1, (2, 3))
```

```
1 (1, [2,3])
```

```
(1, [2, 3])
```

tuples are immutable

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

```
1 x[2] = 5
```

```
TypeError: 'tuple' object does not support item assignment
```

```
1 del x[2]
```

```
TypeError: 'tuple' object doesn't support item deletion
```

```
1 x.clear()
```

```
AttributeError: 'tuple' object has no attribute 'clear'
```

Casting sequences

It is possible to cast between sequence types

```
1 x = [1,2,3]
2 y = (3,2,1)
```

```
1 tuple(x)
```

```
(1, 2, 3)
```

```
1 list(y)
```

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

```
1 tuple(x) == x
```

```
False
```

```
1 list(tuple(x)) == x
```

```
True
```

Ranges

These are the last common sequence type and are a bit special - ranges are a *homogenous*, *ordered*, *immutable* "containers" of **integers**.

```
1 range(10)
```

```
range(0, 10)
```

```
1 list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
1 range(0,10)
```

```
range(0, 10)
```

```
1 list(range(0,10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
1 range(0,10,2)
```

```
range(0, 10, 2)
```

```
1 list(range(0,10,2))
```

```
[0, 2, 4, 6, 8]
```

```
1 range(10,0,-1)
```

```
range(10, 0, -1)
```

```
1 list(range(10,0,-1))
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

What makes ranges special is that `range(1000000)` does not store 1 million integers in memory but rather just three 3*.

Strings as sequences

In most of the ways that count we can think about Python strings as being ordered, immutable, containers of unicode characters and so much of the functionality we just saw can be applied to them.

```
1 x = "abc"
```

```
1 x[0]
```

```
'a'
```

```
1 len(x)
```

```
3
```

```
1 x[-1]
```

```
'c'
```

```
1 "a" in x
```

```
True
```

```
1 x[2:]
```

```
'c'
```

```
1 "bc" in x
```

```
True
```

```
1 x[::-1]
```

```
'cba'
```

```
1 x[0] + x[2]
```

```
'ac'
```

```
1 x[2] = "c"
```

```
TypeError: 'str' object does not support item assignment
```

String Methods

Because string processing is a common and important programming task, the class implements a number of specific methods for these tasks. Review the page linked on the previous slide for help.

```
x = "Hello world! 1234"
```

```
x.find("!")
```

```
11
```

```
x.isalnum()
```

```
False
```

```
x.isascii()
```

```
True
```

```
x.lower()
```

```
'hello world! 1234'
```

```
x.swapcase()
```

```
'hELLO WORLD! 1234'
```

```
x.title()
```

```
'Hello World! 1234'
```

```
x.split(" ")
```

```
['Hello', 'world!', '1234']
```

```
"|".join(x.split(" "))
```

```
'Hello|world!|1234'
```

Find a more complete list of functions here

Exercise 2

String processing - take the string given below and apply the necessary methods to create the target string.

Source:

```
"the quick  Brown   fox Jumped  over   a Lazy  dog"
```

Target:

```
"The quick brown fox jumped over a lazy dog."
```


Hardcoding w/ magic numbers is perfectly acceptable here.

Set and Mapping types

We will discuss sets (`set`) and dictionaries (`dict`) in more detail next week.

Specifically we will discuss the underlying data structure behind these types (as well as lists and tuples) and when it is most appropriate to use each.