

(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 a traditional object oriented class system for implementation

```
1 True
```

```
True
```

```
1 1
```

```
1
```

```
1 1.0
```

```
1.0
```

```
1 1+1j
```

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

```
1 "string"
```

```
'string'
```

```
1 type(True)
```

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

```
1 type(1)
```

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

```
1 type(1.0)
```

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

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

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

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

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

Dynamic types and coercion

As a dynamically typed language, most basic operations will attempt to coerce object to a consistent type appropriate for the operation.

Boolean operations:

```
1 1 and True
```

True

```
1 0 or 1
```

1

```
1 not 0
```

True

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

True

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

False

Comparisons:

```
1 5. > 1
```

True

```
1 5. == 5
```

True

```
1 1 > True
```

False

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

True

```
1 "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

The *default* numeric type in python is an *integer* rather than a *double*, but floats generally take precedence over integers when combined.

Coercion errors

Python is not quite as liberal as R when it comes to type coercion - so operator coercion errors are more common. Do be aware that Python has a number of useful / idiosyncratic overloads for some of the basic operators.

```
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'
```

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

```
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 β = 1 + 2 / 3
2 β
```

1.6666666666666665

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

5

```
1 b
```

5

```
1 a = 3
2 a
```

3

```
1 b
```

5

string literals

Strings can be defined using several 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 and contain quote characters,

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

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

Multiline strings

A single `\` 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"
```

```
'line one not line two 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'
```

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\n cd")
```

ab

cd

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

ab cd

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

ab\\cd

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

ab\n cd

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

ab\tcd

Special values

Base Python does not support missing values. Non-finite floating point values are available but somewhat awkward to use.

```
1 1/0
```

```
ZeroDivisionError: division by zero
```

```
1 1./0
```

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

```
1 nan
```

```
NameError: name 'nan' is not defined
```

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

```
nan
```

```
1 inf
```

```
NameError: name 'inf' is not defined
```

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

```
-inf
```

```
1 5 > float("inf")
```

```
False
```

```
1 5 > float("-inf")
```

```
True
```

There is also a [None](#) type/class which is similar in usage and functionality to [NULL](#) in R.

```
1 None
```

```
1 type(None)
```

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

Sequence types

lists

Python lists are *heterogenous, ordered, mutable* containers of objects (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'>
```

Common operations

```
1 x = [0,1,1,0]
```

```
1 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]

slice steps

```
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, removed, or added to.

```
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]
```

Nested lists

Now lets look at happens when we have a list inside a list and make a change at either level.

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

```
1 x[0] = -1
2 x[1][0] = 5
```

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

```
1 x
```

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

```
1 y
```

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

```
1 z
```

```
[0, [5, 2], [3, 4]]
```

The copy method(s) in Python are generally *shallow* by default. `deepcopy()` is available via the `copy` module - more on this later.

Identity vs equality

To check if two names point to the same object in memory use `is`, to check if they have the same value use `==`. The `id()` function returns the memory address of an object.

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

```
1 x is y
```

True

```
1 x is z
```

False

```
1 x == y
```

True

```
1 x == z
```

True

```
1 id(x)
```

4961988032

```
1 id(y)
```

4961988032

```
1 id(z)
```

4961760832

```
1 id(x) == id(y)
```

True

```
1 id(x) == id(z)
```

False

Value unpacking

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

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

```
1 x
```

```
1
```

```
1 y
```

```
2
```

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

```
1 x
```

```
1
```

```
1 y
```

```
[2, 3]
```

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

```
1 x
```

```
[0, 1]
```

```
1 y
```

```
[2, 3]
```

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

```
1 x1
```

```
0
```

```
1 y1
```

```
1
```

```
1 x2,y2
```

```
(2, 3)
```

Extended unpacking

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

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

```
1 x
```

```
1
```

```
1 y
```

```
[2, 3]
```

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

```
1 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 *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* “container” of **integers**.

```
1 range(10)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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 sequence type functionality we just saw can be applied to them.

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

```
1 x[0]
```

```
'a'
```

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

```
'c'
```

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

```
'c'
```

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

```
'cba'
```

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

```
1 len(x)
```

```
3
```

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

```
True
```

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

```
True
```

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

```
'ac'
```

TypeError: 'str' object does not support item assignment

String Methods

Because string processing is a common and important programming task, the `str` class implements a number of additional methods to support these tasks.

See [here](#) a list of methods.

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

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

11

```
1 x.isalnum()
```

False

```
1 x.isascii()
```

True

```
1 x.lower()
```

'hello world! 1234'

```
1 x.swapcase()
```

'hELLO WORLD! 1234'

```
1 x.title()
```

'Hello World! 1234'

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

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

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

'Hello|world!|1234'

Exercise 2

Take the string given below and apply the necessary methods to create the target string.

Source:

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

Target:

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

Sets and Mapping types

We will discuss sets (`set`) and dictionaries (`dict`) in more detail on Thursday.

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.