Lec 02 - (A very brief) Introduction to Python

Statistical Computing and Computation

Sta 663 | Spring 2022

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).

Some of the core types,

```
True
```

```
## True
```

```
type(True)
```

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

```
1
```

```
## 1
```

```
type(1)
```

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

```
1.0
```

```
## 1.0
```

```
type(1.0)
```

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

```
1+1j
```

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

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

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

Note - all of these types are for 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
```

```
"abc" > 5
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError
```

Mathematical operations

```
1 + 5
```

```
## 6
```

```
1 + 5.
```

```
## 6.0
```

```
1 * 5.
```

```
## 5.0
```

```
True * 5
```

```
## 5
```

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

```
## -1j
```

```
5 / 1.
```

```
## 5.0
```

```
5 / 2
```

```
## 2.5
```

```
5 // 2
```

```
## 2
```

```
5 % 2
```

```
## 1
```

```
7 ** 2
```

```
## 49
```

Math ops and strings

```
"abc" + 5
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: can only concatenate str (not "int") to
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

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

```
## 'abc5'
```

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported operand type(s) for ** or p
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
"abc" * 3
```

```
## 'abcabcabc'
```

Casting

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

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

```
## 0.5
```

```
float(True)
```

```
## 1.0
```

```
int(1.1)
```

```
## 1
```

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

```
## 2
```

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

```
## Error in py_call_impl(callable, dots$args, dots
```

```
bool(0)
```

```
## False
```

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

```
## True
```

```
str(3.14159)
```

```
## '3.14159'
```

```
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.)

```
x = 100
x
```

```
## 100
```

```
x = "hello"
x
```

```
## 'hello'
```

```
ß = 1 + 2 / 3
ß
```

```
## 1.6666666666666665
```

```
a = b = 5
```

```
a
```

```
## 5
```

```
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 names.

string literals

Strings can be defined using a couple of different ways,

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

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

```
"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.

```
"""line one
line two
line three"""
```

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

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 a `None` type which is similar in spirit and functionality to `NULL` in R.

```
1/0
```

```
## Error in py_call_impl(callable, dots$args, dots
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
1./0
```

```
## Error in py_call_impl(callable, dots$args, dots
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

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

```
## nan
```

```
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 down the road

Sequence types

lists

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

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

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

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

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

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

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

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

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

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

Common operations

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

```
2 in x
```

```
## False
```

```
2 not in x
```

```
## True
```

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

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

```
x * 2
```

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

```
len(x)
```

```
## 4
```

```
max(x)
```

```
## 1
```

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

```
## 2
```

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

```
## 0
```

See here and here for a more complete listings.

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 a slice (`start:stop:step`).

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

```
x[0]
```

```
## 1
```

```
x[3]
```

```
## 4
```

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

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

```
x[3:]
```

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

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

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

```
x[:3]
```

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

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

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

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

```
## [1, 4]
```

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

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

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

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

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

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

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

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

mutability

Since lists are mutable the stored values can be changed,

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

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

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

```
del x[0]
x
```

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

```
x.append(7)
x
```

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

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

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

```
x.pop()
```

```
## 7
```

```
x
```

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

```
x.clear()
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.

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

x.append(2)
```

```
x
```

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

```
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`.

```
x = [0,1,1,0]
y = x.copy()

x.append(2)
```

```
x
```

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

```
y
```

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

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

Exercise 1

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

```
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

Value unpacking

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

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

```
## 1
```

```
y
```

```
## 2
```

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

```
## 1
```

```
y
```

```
## [2, 3]
```

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

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

```
y
```

```
## [2, 3]
```

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

```
## 0
```

```
y1
```

```
## 1
```

```
x2
```

Extended unpacking

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

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

```
## 1
```

```
y
```

```
## [2, 3]
```

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

```
## [1, 2]
```

```
y
```

```
## 3
```

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

```
## Error in py_call_impl(callable, dots$args, dots$keywords): ValueError: too many values to unpack (expected 2
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

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 combining multiple objects when returning from a function.

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

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

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

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

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

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

tuples are immutable

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

x[2] = 5
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: 'tuple' object does not support item as
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
del x[2]
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: 'tuple' object doesn't support item del
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
x.clear()
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): AttributeError: 'tuple' object has no attribute 'c
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Casting sequences

It is possible to cast between different sequence types

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

```
tuple(x)
```

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

```
list(y)
```

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

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

```
## False
```

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

```
## True
```

Ranges

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

```
range(10)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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 actually 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.

```
x = "abc"
```

```
x[0]
```

```
## 'a'
```

```
x[-1]
```

```
## 'c'
```

```
x[2:]
```

```
## 'c'
```

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

```
## 'cba'
```

```
len(x)
```

```
## 3
```

```
"a" in x
```

```
## True
```

```
"bc" in x
```

```
## True
```

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

```
## 'ac'
```

String Methods

Because string processing is a common and important programming task, the class implements a number of specific methods for these tasks.

```
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'
```

More complete list here

Exercise 2

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

Source:

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

```
## 'the quick  Brown   fox Jumped  over   a Lazy  dog'
```

Target:

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

```
## '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.