Basics and Data Types

PS 452: Text as Data

Fall 2014

Department of Political Science, Stanford University

Created by **Frances Zlotnick**

Please contact me at zlotnick@stanford.edu with questions or comments.

This document provides some tips for getting started with python and using the basic data types. It assumes you have a working knowledge of R, but no other programming experience.

It covers

- Whitespace
- 0-indexing
- Ints vs Floats
- Booleans
- Strings

Whitespace

In python, whitespace matters. R uses brackets to describe structure, while python uses whitespace. This makes the python easy to read, but if you are copying and pasting from other programs that don't handle spacing consistently, you may run into trouble.

In [1]:
```
for x in range(0,10):
print x
```

```
  File "<ipython-input-1-40ea9d8e6261>", line 2
    print x
        ^
IndentationError: expected an indented block
```

In [2]:
```
for x in range(0,10):
    print x
```

```
0
1
2
3
4
5
6
7
8
9
```

Zero-indexing

Python, like most programming languages, is 0-indexed. That means that the first element in any indexible data type is the 0th element. Indexing element 1 will get you the second element in an object. So remember to start your loops from 0!

In [3]:
```
mylist = ["hello", "this", "is", "a", "list", "of", "strings"]

print mylist[0]
print mylist[1]
```

```
hello
this
```

Data Types

Numeric Types

There are four numeric data types in python, but we will only regularly encounter two: `ints` and `floats`. `ints` are whole numeric values; `floats` are floating point numbers, or values with decimals. This sounds like a trivial difference, but it's not.

In [4]:
```
x = 5
y = 3
x / y
```

Out[4]: 1

In most programming languages, including python versions before 3.0, dividing two `int`s will yield an `int`, but since `int`s can only be whole numbers, any remainder is truncated. The effect is that integer division will give the answer rounded down to the nearest whole number. You can avoid this by casting at least one of the inputs to a `float`.

In [5]:
```
float(x) / y
```

Out[5]:
```
1.6666666666666667
```

Any whole number value will be typed as an int by default, and any value with a decimal with automatically be typed as a float.

In [6]:
```
print type(3)
print type(3.0)
```

```
<type 'int'>
<type 'float'>
```

If your code isn't running or you are getting strange output, check that your data is typed appropriately with the `type` function. Compare that to the documentation for functions you are using; trying to apply a function to a data type it does not accept is one of the most common sources of problems.

In [7]:
```
type(x)
```

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

It's easy enough to cast variables of one type into another, but remember to save your new version if you want the change to persist.

In [8]:
```
x = float(x)
type(x)
```

Out[8]:
```
float
```

Most of the basic arithmetic operations from R work similarly in python. The main differences are integer division, and the notation for exponentiation.

In [9]:
```
print 5 + 2
print 5 - 2
print 5 * 2
print 5 / float(2)   #remember: integer division!
print 5 % 2          #mod operator, returns the remainder of a division operation
print 5**2           #expontiation
```

```
7
3
10
2.5
1
25
```

Booleans

Like R, python has `bool` values that take on values of True or False. These are useful in setting up conditional statements, where an operation is only performed if some condition is satisfied.

In [10]:
```
mybool = True
if (mybool == True):
    print "yes :D"
else:
    print "no :("
```

```
yes :D
```

`bool`s have corresponding numeric values and can be used in integer operations. True = 1 and False = 0.

In [11]:
```
mybool + 1
```

Out[11]:
```
2
```

The equivalence and nonequivalence operators in python are the same as in R, but the conjunction and disjunction operators are simpler. Rather than use & for "and" and | for or, python simply uses the words.

In [12]:
```
mybool == True
```

Out[12]:
```
True
```

In [13]:
```
mybool != True
```

Out[13]:
```
False
```

In [14]:
```
mybool2 = True

if mybool == True and mybool2 == True:
    print "yes!"
else:
    print "no."
```

```
yes!
```

In [15]:
```
if mybool == False or mybool2 == False:
    print "yes!"
else:
    print "no."
```

```
no.
```

Strings

Strings are declared using either double or single quotation marks.

In [16]:
```
a = "hello"
b = 'hello again'
print type(a)
print type(b)
```

```
<type 'str'>
<type 'str'>
```

You can cast the contents of numeric variables to strings using `str`. Many functions, such as `write`, will only accept strings, so you will often need to do this type of casting on numeric variables. Note that you can't always tell whether something is a string just by looking at it.

In [17]:
```
x = 500
y = str(x)
print x
print y
print type(x)
print type(y)
```

```
500
500
<type 'int'>
<type 'str'>
```

Python has many functions for doing common operations on strings. `dir` applied to an object will return a list of the functions that are available for that object. We'll go through just a few of these in this tutorial.

In [18]:
```
dir(y)
```

Out[18]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__getslice__',
 '__gt__',
 '__hash__',
 '__init__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '_formatter_field_name_split',
 '_formatter_parser',
 'capitalize',
 'center',
 'count',
 'decode',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'index',
 'isalnum',
 'isalpha',
 'isdigit',
 'islower',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Strings are immutable: you cannot modify a string by applying a function to it. If you want the changes you make to persist, you must save the new string.

In [19]:
```
fruit = "apple"
print fruit.upper()
print fruit
```

```
APPLE
apple
```

In [20]:
```
fruit = fruit.upper()
print fruit
```

```
APPLE
```

Strings are indexible, which means you can do things like slice them, but remember that 0 indexing applies here as well.

In [21]:
```
fruit[1]
```

Out[21]: 'P'

In [22]:
```
fruit[0:3]
```

Out[22]: 'APP'

You can find the number of characters in a string by applying the `len` function. This will include whitespace and punctuation.

In [23]:
```
len(fruit)
```

Out[23]: 5

To capture the last element in any indexible object, subtract 1 from the length of the object. If you ask for something at an index that doesn't exist in the object, python will throw an error.

In [24]:
```
fruit[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-24-7e4f5eb7330e> in <module>()
----> 1 fruit[5]

IndexError: string index out of range
```

In [25]:
```
fruit[4]
```

Out[25]: 'E'

Strings can be concatenated using the + operator.

In [26]:
```
fruit + " and bananas"
```

Out[26]: 'APPLE and bananas'

Be aware of your data types; the + operator is overloaded, which means it does different things on different data types. On numeric variables, it will perform addition. You can get unexpected behaviors if your variables aren't the type you think they are.

In [27]:
```
x = 5
y = '5'

print x+x
print y+y
```

```
10
55
```

You can split a single string into a list of separate strings at defined delimiters with the `split` function. Keep in mind that none of the resulting strings will contain the delimiter.

In [28]:
```
mystring = "I'd like to split this sentence into a list of words."
mystring.split(" ")
```

Out[28]:
```
["I'd",
 'like',
 'to',
 'split',
 'this',
 'sentence',
 'into',
 'a',
 'list',
 'of',
 'words.']
```

You can strip out specified leading and trailing characters with `strip`.

In [29]:
```
mystring.strip(".")
```

Out[29]: "I'd like to split this sentence into a list of words"

If you don't provide any arguments, `strip` will default to removing whitespace.

In [30]:
```
mystr = "  demonstration of whitespace stripping!!!!    ??      "
mystr.strip()
```

Out[30]: 'demonstration of whitespace stripping!!!!   ??'

To remove characters from the middle of a string, use `replace`. `replace` takes two arguments: the first is the substring you'd like replaced, and the second is what you'd like to replace it with. To remove all instances of a string, set the replacement argument to an empty string.

In [31]:
```
mystr.replace(" ","")
```

Out[31]: 'demonstrationofwhitespacestripping!!!!??'

You can search for a particular substring within a string using `find`. This will return the index of the first character of the first instance of the substring. `rfind` will locate the index of the first character of the last occurence. If the substring does not exist in the searched string, these functions will return a value of -1. The `re` package, discussed in another tutorial, provides more functionality.

In [32]:
```
mystr.find("e")
```

Out[32]: 3

In [33]:
```
mystr.rfind("e")
```

Out[33]: 28

In [34]:
```
mystr.find("x")
```

Out[34]: -1

You can test for various properties of strings using the functions with `is` prefixes. `isalpha` will return a `bool` value of True if all characters in a string are letters (letters), and False if 1 or more characters are not.

In [35]:
```
pres = "OBAMA"
pres2 = "Obama!"
print pres.isalpha()
print pres2.isalpha()
```

```
True
False
```

`isupper` will similarly test whether all of the alphabetic characters are upper case. The analagous `islower` function tests for lower case letters.

In [36]:
```
print pres.isupper()
print pres2.isupper()
```

```
True
False
```

Similar functions exist to test for digits and whitespace.

In [37]:
```
nums1 = "0123456"
nums1.isdigit()
```

Out[37]: True

In [38]:
```
nums2 = "$15.00"
nums2.isdigit()
```

Out[38]: False

In [39]:
```
s = " "
s.isspace()
```

Out[39]: True

`startswith` tests whether a string begins with a substring, provided as an argument.

In [40]:
```
pres.startswith("O")
```

Out[40]: True

But remember that it is case sensitive!

In [41]:
```
pres.startswith("o")
```

Out[41]: False

Summary

Differences from R

- whitespace
- 0 indexing
- integer division

Data Types

ints vs floats

- whole numbers vs decimals
- integer division
- `type`
- + adds values

booleans

- True vs False
- corresponding numeric values of 0 and 1
- ==, !=, and, or

strings

- declare with "" or '', cast with str()
- + concatenates values
- indexible