Cheatsheet

August 25, 2016

1 Python cheat sheet

1.1 Basics

1.1.1 Comments

Single-line comments in Python are denoted by a hash character. You can also make longer comments which can span multiple lines by encasing text in triple quotation marks.

```
In [1]: # This is a comment

        """This is also a comment


        But it is over several lines
        """

Out[1]: 'This is also a comment\n\n\nBut it is over several lines\n'
```

1.1.2 Printing

Printing in Python is done with the `print()` function.

```
In [2]: print("Hello World!")

Hello World!
```

The `print()` can print multiple things at once by separating them with commas. Strings can include escape characters such as `\t` (tab) and `\n` (newline).

```
In [3]: print("\tThis is a bigger print statement over",
              2, "lines.\nIsn't it great?")

        This is a bigger print statement over 2 lines.
Isn't it great?
```

1.1.3 Variables and operators

Variables are defined as you would expect, but note that unlike Fortran and other low-level languages, you do not need to declare variables before defining them.

```
In [4]: # Define an integer
        x = 1

        # Define a float
        x = 3.14

        # Define a string in different ways
        x = 'This is a string'
        y = "This is also a string"
        z = """This is a long string.
        Just like the comment earlier."""

        # Define some booleans
        x = True
        y = False
```

Basic operators are also as you would expect (+, −, *, /, **). When performing an operation with variables of different types, Python will try to cast one to the other's type if appropriate. Otherwise it will fail and raise an `Exception` (a runtime error).

```
In [5]: print(1 + 3)
        print(1 + 3.14)
        print(1 + '3')
```

```
4
4.140000000000001



        ------------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-5-c4afba7f1571> in <module>()
          1 print(1 + 3)
          2 print(1 + 3.14)
    ----> 3 print(1 + '3')


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

1.2 Sequences

1.2.1 Lists

The basic sequence type is the *list*, which is defined with square brackets around comma-separated items. Each item in a list can have any type - they don't all need to be the same. As well as allowing you to group different types together in a sequence, this also allows you to nest another sequence within your list.

Lists are *mutable*, which is to say that once defined, the list is not fixed forever - you can add or remove items as you please.

```
In [6]: # Define some lists
        x = [1, 2, 3]
        y = ['one', 'two', 'three']
        z = ['string', 1, [4.2, 'another string']]

        # Add to a list
        print(z)
        z.append('new item')
        print(z)

        # Remove from a list
        z.remove('string')
        print(z)

        # Concatenate two lists
        print(x + y)
```

```
['string', 1, [4.2, 'another string']]
['string', 1, [4.2, 'another string'], 'new item']
[1, [4.2, 'another string'], 'new item']
[1, 2, 3, 'one', 'two', 'three']
```

1.2.2 Indices and slicing

Sequences are indexed with square brackets, and indices start at 0. Nested sequences are indexed with separate brackets.

You can slice from one index position to another by putting a colon between indices. Note that when doing this, the result is not inclusive of the end index. Putting another integer after a second colon will specify the increment between items in the slice. With this syntax, you can omit the start index, which will then default to 0, or the end index, which will default to the end of the sequence.

You can also index sequences backwards with negative indices. Note that in this case the last element is at index -1, not 0. Similarly, increments in slices can be negative, causing the items in the resulting slice to be backwards compared to the original list.

```
In [7]: # Get the first item of a list
        print(x[0])
```

```
1
```

```
In [8]: # Reference a list in a list
        a = z[-2][1]
        print(a)
```

```
another string
```

```
In [9]: # Slice a list from the second element to the last
        print(y[1:3])
        # Or, equivalently
        print(y[1:])
```

```
['two', 'three']
['two', 'three']
```

```
In [10]: # Slice a list from the second-to-last element to the last
         print(z[-2:])
```

```
[[4.2, 'another string'], 'new item']
```

```
In [11]: # Index with an increment
         print(z[0:-1:2])
         # Or, equivalently
         print(z[::2])
```

```
[1]
[1, 'new item']
```

```
In [12]: # Index backwards
         print(z[::-1])
```

```
['new item', [4.2, 'another string'], 1]
```

1.2.3 Other sequences

Python treats strings as a kind of sequence and as such they are indexed in the same way as lists.

```
In [13]: print(a)
         # Index a string
         print(a[6])
```

```
another string
r
```

Another useful type of sequence in Python is the *dictionary*, or *dict*. This is an unordered grouping of (*key*, *value*) pairs. Both the key and the value can be any type. Values are accessed similarly to items in lists, but by specifying the key rather than an index.

```
In [14]: # Define a dict
         x = {'first number': 3.14, 'second number': 42,
              'third number': 'something else'}

         # Print the dict to note that the entries
         # are not stored in the order we specified
         print(x)

         # Get the value associated with the key 'second number'
         print(x['second number'])
```

```
{'second number': 42, 'first number': 3.14, 'third number': 'something else'}
42
```

1.3 Logic

Comparison operators work in the same way as in most other languages.

```
In [15]: # Compare some variables
         x = 3
         y = 5

         print(x < y)  # Less than
         print(x > y)  # Greater than
         print(x <= y)  # Less than or equal to
         print(x >= y)  # Greater than or equal to
         print(x == y)  # Equal to
         print(x != y)  # Not equal to
```

```
True
False
True
False
False
True
```

`if` statements consist of the `if` keyword followed by a condition and a colon, then an indented block of code to run if the condition is met. Note that the indentation is not just aesthetic - it is part of the syntax and an improperly indented if statement will raise an `IndentationError`.

Blocks can be indented by any amount, as long as it is the same for every line within any one block, but the convention is four spaces (in a Jupyter notebook, the tab key will insert four spaces rather than a tab character). Since the extent of the `if` block is defined by the indentation, there is no ‘end if’ in Python.

Optionally, `if` blocks can include one or more `elif` (short for 'else if') clause and/or an `else` clause.

```
In [16]: # A basic if block with multiple clauses
         if x > y:
             print('x is larger')
         elif x < y:
             print('y is smaller')
         else:
             print('x and y are equal')
```

```
y is smaller
```

1.4 Loops

1.4.1 `for` loops

To loop over a sequence in Python, we use the `for` keyword. The syntax for this is similar to that for `if` statements:

```
In [ ]: # For loop syntax
        for item in sequence:
            do some things
            in an indented block
```

Remember that the indentation is

Note that what Python is actually doing here is not quite the same as loops in most other languages. Rather than iterating over a range of numbers that you use to index a sequence (although you can do this) it is actually iterating directly over the items in the specified sequence.

```
In [17]: # Loop over a list
         for x in [0, 'one', 2.0, [3, '.14'], 4]:
             print(x)
```

```
0
one
2.0
[3, '.14']
4
```

1.4.2 `while` loops

Python also has a `while` loop for repeating code until a condition is met. Again, the syntax for this is similar to what we've seen already.

```
In [18]: x = 0

         while x < 5:
             print(x)
             x = x + 1
```

```
0
1
2
3
4
```

1.5 Functions

Syntax for function definitions is again similar - statement, colon, indented block. The definition is specified by a `def` keyword, followed by the name of the function, then the arguments to the function in brackets and separated by commas.

The `return` statement allows you to specify the output of the function. Without a `return` statement, the default output of a function is `None`, which is a null variable.

```
In [19]: # Define a function with an argument
         def times(x, y):
             """Multiply two numbers, x and y."""
             return x * y

         # Call the function
         x = times(3, 3)
         print(x)
```

```
9
```

It's important to notice here that there are different definitions of x inside and outside the function. This is because Python separates variables into *namespaces*, to reduce the risk of variables being accidentally overwritten. The function has its own namespace, and only knows about the variables defined there or passed in from outside.

As well as normal arguments, functions can take *keyword arguments*, or *kwargs*. These define default values for the argument and can be passed to a function in any order using the keyword.

```
In [20]: # Define a new function with a kwarg
         def topower(x, power=2):
             """Raise a number x to power."""
             return x ** power

         # Three ways of calling the function
         print(topower(x), topower(x, 3), topower(x, power=3))
```

```
81 729 729
```

```
In [21]: # Define a function with multiple kwargs
         def topower(x, power=2, root=False):
             """Raise a number x to a power, or find the power-th root."""
             if root:
```

```
            y = x ** (1 / power)
        else:
            y = x ** power

        return y

    # Kwargs can be used individually or together,
    # and in any order (as long as the keyword is used)
    print(topower(9),
          topower(9, 3, True),
          topower(9, root=True, power=3),
          topower(x, root=True))
```

```
81 2.080083823051904 2.080083823051904 3.0
```