A quick guide to python

Roberto Castro Sundin

2019-03-20

Introduction

This guide is intended as a quickstart guide for people new to python, but with previous experience with some other programming language. The explanations are kept short intendedly and each subject is covered in very little detail, with the means of keeping the guide short, yet provide the tools needed to get started. My suggestions are to read the guide while having an open python console closeby, in order to check the results given in the examples and to do further experiments on your own. When this guide isn't enough I recommend you to refer to the official documentation.

A note on python versions

This guide is written with python 3.7 kept in mind. Please note that python 2 is officially considered a dead language as of 2020. I therefore *strongly* advice you to use python 3+.

Help

A very useful function is the `help()` function. Whenever in doubt, use it. Help on any built-in function `function()` can be found by typing `help(function)` into the python console (note that no parentheses should be used for the function you want help for).

Data types

The available data types in python are

- Integers
- Floats
- Complex numbers
- Booleans
- Strings
- Lists/Arrays
- Tuples
- Dictionaries
- Sets

Most should be familiar with the upper 6 data types, whilst not knowing too much about the bottom 3, as they are more somewhat specific to the python language. We will not go into too

much detail about what these are other than that they are types of arrays with the following exceptions/properties

- Tuples
  - Like lists but *immutable* (cannot be changed once defined)
- Dictionaries
  - Hashed, have ordered pairs of *keywords* and their respective *values*.
- Sets
  - Just like mathematical sets, no duplicates allowed.

Variable declaration; duck-typing

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck.

Python uses what is commonly referred to as *duck-typing*, meaning that instead of explicitly declaring the data type of a variable, python will set the data type according to how the variable is defined. Let's have a look at this by creating two variables `a` and `b`:

```
a = 1.0
b = 1
```

If we now look at what type each variable is, we can see that `a` is of type "float"

```
print(type(a))
```

```
## <type 'float'>
```

and that `b` is of type "int"

```
print(type(b))
```

```
## <type 'int'>
```

The interpreter has thus classified `a` as a float, because of the decimal point in the declaration and `b` as an integer because of a lack thereof.

Benefits of duck-typing

Duck-typing does not only allow for easy creation of variables, it also comes in handy when dealing with several variables and performing operations between them. An example of this is that arithmetic operations between ints, floats and complex numbers is perfectly legit in python

```
a = 1 # python creates an int
# Lets add a float
a += 2.0
print(type(a),a)
```

```
## (<type 'float'>, 3.0)
```

By adding a float to our int, python first converted the int to float and then performed the addition, resulting in a float with a value of 3.0.

Illegal operations

Our previous example might insinuate that python is some kind of free-for-all programming language where everything is allowed. This, however, is not the case. Less common sense operations like adding an int to a string throws an exception

```
'hello world'+1
```

```
## TypeError: cannot concatenate 'str' and 'int' objects
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

The answer on how to do this correctly might depend on the desired outcome but will be described in the following section.

Converting between data types

Converting between data types is fairly straightforward in python and is done using built-in functions. Assuming `x` is a variable of a data type compatible for conversion, these are the possible built-in functions that can be used

- `int(x)`
- `float(x)`
- `complex(x)`
- `bool(x)`
- `str(x)`
- `list(x)`
- `tuple(x)`
- `dict(x)`
- `set(x)`

Again, not all operations are allowed, e.g conversion from `list` to `int`. Use common sense, and when in doubt use `help()`, e.g `help(str)`.

Knowing this, the error from our previous example can be avoided

```
a = 'hello world'
b = 1
print(
a+str(b)
)
```

```
## hello world1
```

Quickly about data types

Ints

- Creation: `a = x`, where `x` is an integer.

Floats

- Creation: `a = x.y`, where `x`, `y` are integers.

Complex

- Creation: e.g `a = 2.0+3j`.

Floats

- Creation: e.g `a = True`.

Strings

- Creation: `a = 'hello world'`.

Lists

- Creation: `a = [1,2,3]`.

Lists are indexed and the *i*:th element is fetched using brackets, e.g `a[i]`

```
print(a[0])
```

```
## 1
```

The elements can also be changed the same way

```
a[1] = 10
print(a)
```

```
## [1, 10, 3]
```

Tuples

- Creation: `a = (1,2,3)`.

Tuples are indexed and values are fetched just as for lists. However, once created, elements cannot be changed (tuples are immutable!)

```
a[1] = 10
```

```
## TypeError: 'tuple' object does not support item assignment
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Dictionaries

- Creation: `a = {'one': 1, 'two': 2}`.

Values can be fetched by using the `get()` method

```
print(a.get('one'))
```

```
## 1
```

Dictionary keywords can also be run through using `for in`-loops

```
for key in a:
    print(key)
```

```
## two
## one
```

Sets

- Creation: `a = {1,2,3}`.

Control statements

Logical operators

The following logical operators can be used in python

Sign	Description
`==`	equals
`!=`	not equal
`>=`	bigger or equal
`<=`	smaller or equal
`>`	bigger than
`<`	smaller than
`not`	not
`and`	and
`or`	or
`is`	is

If-statements

Just as in other programming languages, the `if`-statement in python allows the programmer to specify a code chunk that will be run if and only if a certain condition is fulfilled. Python uses `if`, `elif`, and `else` statements and the syntax is the following

```
a = 1
b = 2.0
condition1 = a < b
condition2 = a > b
if condition1:
```

```
    print('a < b')
elif condition2:
    print('a > b')
else:
    print('a = b')
```

```
## a < b
```

The flow scheme of the `if`-statement is the following

- The program evaluates `condition1`
  - If `condition1` is `True`, the program runs the indented code-block under the statement and then skips *both* the `elif` and the `else`-statement, without even evaluating `condition2`.
  - If `condition1` is `False`, the program proceeds to evaluate `condition2`.
    * If `condition2` is `True`, the program runs the indented code-block under the `elif` statement and then skips `else`-statement.
    * If `condition2` is `False`, the program runs the indented code-block under the `else` statement.

While-loops

The syntax for while-loops in python is the following:

```
a = 0
list = []
while a<5:
    list.append(a)
    a += 1
print(list)
```

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

For-loops

The syntax for while-loops in python is the following:

```
list = []
for i in range(5):
```

```
    list.append(i)
print(list)
```

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

If the index used for counting is not used in the for loop, the convention is to use an underscore in the following way

```
a = 0
for _ in range(5):
    a += 1
print(a)
```

```
## 5
```

A very *pythonic* feature is the possibility to run through e.g all elements of a list

```
fruits = ['apple','banana','orange']
for fruit in fruits:
    print(fruit)
```

```
## apple
## banana
## orange
```

This is to be contrasted with the clumsier approach

```
for i in range(len(fruits)):
    print(fruits[i])
```

```
## apple
## banana
## orange
```

Functions

Functions are defined in the following way in python

```
def function(x,y,z):
    b = x+y+z
    return b
```

Here `x`, `y` and `z` are inputs, which again, because of the duck-typing standard, are not strictly defined data types. We can now use the function by making a function call `function(x,y,z)`

```
print(
function(1,1,1)
)
```

```
## 3
```

The duck-typing convention allows us to use the function with any inputs that can be summed together in python, thus also with strings

```
print(
function('hello','my','friend')
)
```

```
## hellomyfriend
```

Although very handy, this functionality must be used with great care. Or rather, it must be *very clear* for a potential user what is allowed to be sent in as inputs. Also, the programmer must assure that what is being sent in is actually of the intended data type, since the python interpreter itself will not complain as long as all operations in the functions are allowed.

Classes

Defining the class

Classes are defined in the following way in python

```
class Person():
    def __init__(self):
        self.type = 'Person'
```

The `__init__()` function is a constructor and is run each time an object of the class is instantiated.

An object is created the following way

```
a = Person()
```

and to see the effect of the constructor:

```
print(
a.type
)
```

```
## Person
```

Attributes can be accessed and changed

```
a.type = 'Tiger'
print(a.type)
```

```
## Tiger
```

New attributes can also be added

```
a.name = 'John Doe'
print(a.name)
```

```
## John Doe
```

We can also choose to create attributes based on inputs given upon creating the class. This is done the following way

```
class Person():
    def __init__(self, name):
        self.name = name
        self.type = 'Person'
```

Now a input is required upon instantiating an object

```
a = Person('John Doe')
print(a.name)
```

```
## John Doe
```

Methods

Methods are defined in the following way

```
class Person():
    def __init__(self, name):
        self.name = name
        self.type = 'Person'
    def change_name(self, new_name):
        self.name = new_name
```

and are run as follows

```
a.change_name('Jan Daw')
print(a.name)
```

```
## Jan Daw
```

Note that all methods must include `self` as the first argument.