
A quick guide to python

Roberto Castro Sundin

2019-03-20

Contents

1. Introduction	3
1.1. A note on python versions	3
1.2. Help	3
2. Data types	3
2.1. Variable declaration; duck-typing	4
2.1.1. Benefits of duck-typing	4
2.1.2. Illegal operations	5
2.2. Converting between data types	5
2.3. Quickly about data types	6
2.3.1. Ints	6
2.3.2. Floats	6
2.3.3. Complex	6
2.3.4. Floats	6
2.3.5. Strings	6
2.3.6. Lists	7
2.3.7. Tuples	7
2.3.8. Dictionaries	8
2.3.9. Sets	8
3. Control statements	8
3.1. Logical operators	8
3.2. If-statements	9
3.3. While-loops	10
3.4. For-loops	10
4. Functions	11
5. Classes	12
5.1. Defining the class	12
5.2. Methods	13
6. Pure Python Power (Pythonic Possibilities)	13
6.1. List comprehensions	13
A. Some python conventions	15
A.1. Variable names	15

A.2. Class names	15
A.3. Indentation	15

1. 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](#).

1.1. 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+.

1.2. 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).

2. 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.

2.1. 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.

2.1.1. 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.

2.1.2. 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.

2.2. 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
```

2.3. Quickly about data types

2.3.1. Ints

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

2.3.2. Floats

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

2.3.3. Complex

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

2.3.4. Booleans

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

2.3.5. Strings

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

2.3.6. Lists

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

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

```
a = [1,2,3]
print(a[0])
```

```
## 1
```

The elements can also be changed the same way

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

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

A very pythonic feature is that negative numbers can be used to count backwards. The last element of a list can thus be accessed the following way

```
print(a[-1])
```

```
## 3
```

We can also *slice* a list using brackets and a colon. For example, `a[x:y]` returns a list containing the elements of a list `a` with index `x` through `y-1`

```
print(a[1:4])
```

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

Omitting `x`, returns a list starting from the first element until index `y-1`, while omitting `y` returns a list starting from index `x` until the end of the list. Omitting both returns the original list.

Note that slicing does not alter the original list, but only returns a new list that can be assigned to a variable.

2.3.7. 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 = (1,2,3)
a[1] = 10
```

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

2.3.8. Dictionaries

- Creation: `a = {'keyword1': 1, 'keyword2': 'two'}`.

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

```
a = {'keyword1': 1, 'keyword2': 'two'}
print(a.get('keyword2'))
```

```
## two
```

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

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

```
## keyword2
## keyword1
```

Note that keywords are not run through in the same order as they were declared.

2.3.9. Sets

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

3. Control statements

3.1. 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

3.2. 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.

3.3. 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]
```

3.4. For-loops

The syntax for for-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 explicitly stated inside 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, for example, 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
```

4. 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.

5. Classes

5.1. 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 an input is required upon instantiating an object

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

```
## John Doe
```

5.2. 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.

6. Pure Python Power (Pythonic Possibilities)

6.1. List comprehensions

Oftentimes we need to create arrays containing certain type of elements. Let's say for example that we want a list containing numbers from 0 to 9. A standard approach would be to create an empty list and appending these numbers, one-by-one, in a for-loop

```
a = []
for i in range(10):
    a.append(i)
print(a)
```

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

In python, we can harness the power of so called *list comprehensions* to fit this into one single line

```
a = [i for i in range(10)]
print(a)
```

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

Let's say we now want all even numbers from 2 to 18. Either we can make a slicing operation

```
a = [2*i for i in range(10)][1:]
print(a)
```

```
## [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

or adjust the input of the range-function

```
a = [2*i for i in range(1,10)]
print(a)
```

```
## [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

but perhaps the most clean and easily read solution, is to use feature of list comprehensions called *filtering*

```
a = [i for i in range(20) if i%2==0 and i!=0]
print(a)
```

```
## [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

where the filtering part is coming from the if-statement added at the end.

A. Some python conventions

The python language should preferably be written under the guidelines of [PEP8](#), written partly by python creator *Guido van Rossum*.

A.1. Variable names

For variable naming use `lower_case_with_underscores`

A.2. Class names

Classes use the `CapitalizedWords` convention.

A.3. Indentation

Don't use tab, rather use 4 spaces.