1.2.2. Basic types

1.2.2.1. Numerical types

Python supports the following numerical, scalar types:

Integer:

```
>>> 1 + 1
2
>>> a = 4
>>> type(a)
<type 'int'>
```

Floats:

```
>>> c = 2.1
>>> type(c)
<type 'float'>
```

Complex:

```
>>> a = 1.5 + 0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> type(1. + 0j)
<type 'complex'>
```

Booleans:

```
>>> 3 > 4
False
>>> test = (3 > 4)
```

```
>>> test
False
>>> type(test)
<type 'bool'>
```

A Python shell can therefore replace your pocket calculator, with the basic arithmetic operations `+`, `-`, `*`, `/`, `%` (modulo) natively implemented

```
>>> 7 * 3.
21.0
>>> 2**10
1024
>>> 8 % 3
2
```

Type conversion (casting):

```
>>> float(1)
1.0
```

Integer division

In Python 2:

```
>>> 3 / 2
1
```

In Python 3:

```
>>> 3 / 2
1.5
```

To be safe: use floats:

```
>>> 3 / 2.
1.5
```

```
>>> a = 3
>>> b = 2
>>> a / b # In Python 2
1
>>> a / float(b)
1.5
```

Future behavior: to always get the behavior of Python3

```
>>> from __future__ import division
>>> 3 / 2
1.5
```

If you explicitly want integer division use `//`:

```
>>> 3.0 // 2
1.0
```

The behaviour of the division operator has changed in Python 3.

1.2.2.2. Containers

Python provides many efficient types of containers, in which collections of objects can be stored.

1.2.2.2.1. Lists

A list is an ordered collection of objects, that may have different types. For example:

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> type(colors)
```

```
<type 'list'>
```

Indexing: accessing individual objects contained in the list:

```
>>> colors[2]
'green'
```

Counting from the end with negative indices:

```
>>> colors[-1]
'white'
>>> colors[-2]
'black'
```

⚠ **Indexing starts at 0** (as in C), not at 1 (as in Fortran or Matlab)!

Slicing: obtaining sublists of regularly-spaced elements:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[2:4]
['green', 'black']
```

⚠ Note that `colors[start:stop]` contains the elements with indices `i` such as `start<= i < stop` (`i` ranging from `start` to `stop-1`). Therefore, `colors[start:stop]` has `(stop - start)` elements.

Slicing syntax: `colors[start:stop:stride]`

All slicing parameters are optional:

```
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors[3:]
['black', 'white']
```

```
>>> colors[:3]
['red', 'blue', 'green']
>>> colors[::2]
['red', 'green', 'white']
```

Lists are *mutable* objects and can be modified:

```
>>> colors[0] = 'yellow'
>>> colors
['yellow', 'blue', 'green', 'black', 'white']
>>> colors[2:4] = ['gray', 'purple']
>>> colors
['yellow', 'blue', 'gray', 'purple', 'white']
```

Note: The elements of a list may have different types:

```
>>> colors = [3, -200, 'hello']
>>> colors
[3, -200, 'hello']
>>> colors[1], colors[2]
(-200, 'hello')
```

For collections of numerical data that all have the same type, it is often **more efficient** to use the `array` type provided by the `numpy` module. A NumPy array is a chunk of memory containing fixed-sized items. With NumPy arrays, operations on elements can be faster because elements are regularly spaced in memory and more operations are performed through specialized C functions instead of Python loops.

Python offers a large panel of functions to modify lists, or query them. Here are a few examples; for more details, see https://docs.python.org/tutorial/datastructures.html#more-on-lists

Add and remove elements:

```
>>> colors = ['red', 'blue', 'green', 'black', 'white']
>>> colors.append('pink')
```

```
>>> colors
['red', 'blue', 'green', 'black', 'white', 'pink']
>>> colors.pop() # removes and returns the last item
'pink'
>>> colors
['red', 'blue', 'green', 'black', 'white']
>>> colors.extend(['pink', 'purple']) # extend colors, in-place
>>> colors
['red', 'blue', 'green', 'black', 'white', 'pink', 'purple']
>>> colors = colors[:-2]
>>> colors
['red', 'blue', 'green', 'black', 'white']
```

Reverse:

```
>>> rcolors = colors[::-1]
>>> rcolors
['white', 'black', 'green', 'blue', 'red']
>>> rcolors2 = list(colors)
>>> rcolors2
['red', 'blue', 'green', 'black', 'white']
>>> rcolors2.reverse() # in-place
>>> rcolors2
['white', 'black', 'green', 'blue', 'red']
```

Concatenate and repeat lists:

```
>>> rcolors + colors
['white', 'black', 'green', 'blue', 'red', 'red', 'blue', 'green', 'black',
      'white']
>>> rcolors * 2
['white', 'black', 'green', 'blue', 'red', 'white', 'black', 'green',
      'blue', 'red']
```

Sort:

```
>>> sorted(rcolors) # new object
['black', 'blue', 'green', 'red', 'white']
>>> rcolors
['white', 'black', 'green', 'blue', 'red']
>>> rcolors.sort()  # in-place
>>> rcolors
['black', 'blue', 'green', 'red', 'white']
```

Methods and Object-Oriented Programming

The notation `rcolors.method()` (e.g. `rcolors.append(3)` and `colors.pop()`) is our first example of object-oriented programming (OOP). Being a `list`, the object *rcolors* owns the *method function* that is called using the notation **.**. No further knowledge of OOP than understanding the notation **.** is necessary for going through this tutorial.

Discovering methods:

Reminder: in Ipython: tab-completion (press tab)

```
In [28]: rcolors.<TAB>
rcolors.__add__           rcolors.__iadd__          rcolors.__setattr__
rcolors.__class__         rcolors.__imul__          rcolors.__setitem__
rcolors.__contains__      rcolors.__init__          rcolors.__setslice__
rcolors.__delattr__       rcolors.__iter__          rcolors.__sizeof__
rcolors.__delitem__       rcolors.__le__            rcolors.__str__
rcolors.__delslice__      rcolors.__len__
      rcolors.__subclasshook__
rcolors.__doc__           rcolors.__lt__            rcolors.append
rcolors.__eq__            rcolors.__mul__           rcolors.count
rcolors.__format__        rcolors.__ne__            rcolors.extend
rcolors.__ge__            rcolors.__new__           rcolors.index
rcolors.__getattribute__  rcolors.__reduce__        rcolors.insert
rcolors.__getitem__       rcolors.__reduce_ex__     rcolors.pop
```

```
rcolors.__getslice__       rcolors.__repr__          rcolors.remove
rcolors.__gt__             rcolors.__reversed__      rcolors.reverse
rcolors.__hash__           rcolors.__rmul__          rcolors.sort
```

1.2.2.2.2. Strings

Different string syntaxes (simple, double or triple quotes):

```
s = 'Hello, how are you?'
s = "Hi, what's up"
s = '''Hello,                 # tripling the quotes allows the
       how are you'''         # the string to span more than one line
s = """Hi,
what's up?"""
```

```
In [1]: 'Hi, what's up?'
------------------------------------------------------------
   File "<ipython console>", line 1
    'Hi, what's up?'
           ^
SyntaxError: invalid syntax
```

The newline character is `\n`, and the tab character is `\t`.

Strings are collections like lists. Hence they can be indexed and sliced, using the same syntax and rules.

Indexing:

```
>>> a = "hello"
>>> a[0]
'h'
>>> a[1]
'e'
```

```
>>> a[-1]
'o'
```

(Remember that negative indices correspond to counting from the right end.)

Slicing:

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,'
>>> a[2:10:2] # Syntax: a[start:stop:step]
'lo o'
>>> a[::3] # every three characters, from beginning to end
'hl r!'
```

Accents and special characters can also be handled in Unicode strings (see https://docs.python.org/tutorial/introduction.html#unicode-strings).

A string is an **immutable object** and it is not possible to modify its contents. One may however create new strings from the original one.

```
In [53]: a = "hello, world!"
In [54]: a[2] = 'z'
---------------------------------------------------------------------------
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

In [55]: a.replace('l', 'z', 1)
Out[55]: 'hezlo, world!'
In [56]: a.replace('l', 'z')
Out[56]: 'hezzo, worzd!'
```

Strings have many useful methods, such as `a.replace` as seen above. Remember the `a.` object-oriented notation and use tab completion or `help(str)` to search for new methods.

See also: Python offers advanced possibilities for manipulating strings, looking for patterns or formatting. The interested reader is referred to https://docs.python.org/library/stdtypes.html#string-methods and https://docs.python.org/library/string.html#new-string-formatting

String formatting:

```
>>> 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string')
'An integer: 1; a float: 0.100000; another string: string'

>>> i = 102
>>> filename = 'processing_of_dataset_%d.txt' % i
>>> filename
'processing_of_dataset_102.txt'
```

1.2.2.2.3. Dictionaries

A dictionary is basically an efficient table that **maps keys to values**. It is an **unordered** container

```
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'sebastian': 5578, 'francis': 5915, 'emmanuelle': 5752}
>>> tel['sebastian']
5578
>>> tel.keys()
['sebastian', 'francis', 'emmanuelle']
>>> tel.values()
[5578, 5915, 5752]
>>> 'francis' in tel
True
```

It can be used to conveniently store and retrieve values associated with a name (a string for a date, a name, etc.). See https://docs.python.org/tutorial/datastructures.html#dictionaries for more information.

A dictionary can have keys (resp. values) with different types:

```
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
{'a': 1, 3: 'hello', 'b': 2}
```

1.2.2.2.4. More container types

Tuples

Tuples are basically immutable lists. The elements of a tuple are written between parentheses, or just separated by commas:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

Sets: unordered, unique items:

```
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
set(['a', 'c', 'b'])
>>> s.difference(('a', 'b'))
set(['c'])
```

1.2.2.3. Assignment operator

Python library reference says:

> Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects.

In short, it works as follows (simple assignment):

1. an expression on the right hand side is evaluated, the corresponding object is created/obtained
2. a **name** on the left hand side is assigned, or bound, to the r.h.s. object

Things to note:

- a single object can have several names bound to it:

```
In [1]: a = [1, 2, 3]
In [2]: b = a
In [3]: a
Out[3]: [1, 2, 3]
In [4]: b
Out[4]: [1, 2, 3]
In [5]: a is b
Out[5]: True
In [6]: b[1] = 'hi!'
In [7]: a
Out[7]: [1, 'hi!', 3]
```

- to change a list *in place*, use indexing/slices:

```
In [1]: a = [1, 2, 3]
In [3]: a
Out[3]: [1, 2, 3]
In [4]: a = ['a', 'b', 'c'] # Creates another object.
In [5]: a
```

```
Out[5]: ['a', 'b', 'c']
In [6]: id(a)
Out[6]: 138641676
In [7]: a[:] = [1, 2, 3] # Modifies object in place.
In [8]: a
Out[8]: [1, 2, 3]
In [9]: id(a)
Out[9]: 138641676 # Same as in Out[6], yours will differ...
```

- the key concept here is **mutable vs. immutable**
    - mutable objects can be changed in place
    - immutable objects cannot be modified once created

See also: A very good and detailed explanation of the above issues can be found in David M. Beazley's article Types and Objects in Python.