Unit 2: Language and NumPy basics

Richard Foltyn

November 14, 2020

Contents

1 Unit 2: Language and NumPy basics

In this unit we start exploring the Python language, covering the following topics:

1. Basic syntax
2. Built-in data types
3. NumPy arrays

1.1 Basic syntax

- Everything after a # character (until the end of the line) is a comment and will be ignored.
- Variable names are case sensitive.
- Whitespace characters matter (unlike in most languages)!
- Python uses indentation (usually 4 spaces) to group statements, for example loop bodies, functions, etc.
- You don't need to add a character to terminate a line, unlike in some languages.
- You can use the `print()` function to inspect almost any object.

[1]:
```
# First example

# create a variable named 'text' that stores the string 'Hello, world!'
text = 'Hello, world!'

# print contents of 'text'
print(text)
```

```
Hello, world!
```

In IPython notebooks and interactive command-line environments, we can also display a value by simply writing the variable name.

[2]:
```
text
```

[2]:
```
'Hello, world!'
```

This does not do anything in *proper* python script files that are run through the interpreter, though.

1.2 Built-in data types

Pythons is a dynamically-typed language:

- Unlike in C or Fortran, you don't need to declare a variable or its type

You can inspect a variable's type using the built-in `type()` function.

We now look at the most useful built-in data types:

Basic types

- integers (`int`)
- floating-point numbers (`float`)
- strings (`str`)

Containers (or collections)

- tuples (`tuple`)
- lists (`list`)
- dictionaries (`dict`)

1.2.1 Integers and floats

[3]:
```
# Integer variables
i = 1
type(i)
```

[3]:
```
int
```

[4]:
```
# Floating-point variables
x = 1.0
type(x)
```

[4]:
```
float
```

[5]:
```
# A name can reference any data type:
# Previously, x was a float, now it's an integer!
x = 1
type(x)
```

[5]:
```
int
```

It is good programming practice to specify floating-point literals using a decimal point. It makes a difference in a few cases (especially when using NumPy arrays, Numba or Cython):

[6]:
```
x = 1.0 # instead of x = 1
```

1.2.2 Strings

The string data type stores sequences of characters.

[7]:
```
# Strings need to be surrounded by single (') or double (") quotes!
institution = 'University of Glasgow'
institution = "University of Glasgow"
```

1.2.3 Tuples

Tuples represent a collection of several items which can have different data types. They are created whenever several items are separated by commas. The parenthesis are optional:

```
(item1, item2, ...)
```

[8]:
```
# A tuple containing a string, an integer and a float
items = ('foo', 1, 1.0)
items
```

[8]:
```
('foo', 1, 1.0)
```

The parenthesis are optional, but improve readability:

[9]:
```
items = 'foo', 1, 1.0 # equivalent way to create a tuple
items
```

[9]:
```
('foo', 1, 1.0)
```

We use brackets `[]` to access an element in a tuple (or any other container object)

[10]:
```
first = items[0] # contains 'foo'
first
```

[10]:
```
'foo'
```

Python indices are 0-based, so 0 references the first element, 1 the second element, etc.

[11]:
```
second = items[1] # contains second element
second
```

[11]:
```
1
```

Tuples are immutable, which means that the items stored in the tuple cannot be changed!

[12]:
```
# This raises an error!
items = 'foo', 1, 1.0
items[0] = 123
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-b4be5dad43c1> in <module>
      1 # This raises an error!
      2 items = 'foo', 1, 1.0
----> 3 items[0] = 123

TypeError: 'tuple' object does not support item assignment
```

1.2.4 Lists

Lists are like tuples, except that they can be modified. We create lists using brackets,

```
[item1, item2, ...]
```

[13]:
```
# Create list contains a string, an integer and a float
lst = ['foo', 1, 1.0]
lst
```

[13]:
```
['foo', 1, 1.0]
```

Accessing list items works the same as with tuples

[14]:
```
lst[0]      # print first item
```

[14]:
```
'foo'
```

Lists items can be modified:

[15]:
```
lst[0] = 'bar'
lst
```

[15]:
```
['bar', 1, 1.0]
```

Lists are full-fledged objects that support various operations, for example

[16]:
```
lst.insert(0, 'abc')     # insert element at position 0
lst.append(2.0)          # append element at the end
del lst[3]               # delete the 4th element
lst
```

[16]:
```
['abc', 'bar', 1, 2.0]
```

The built-in functions `len()` returns the number of elements in a list (and any other container object)

[17]:
```
n = len(lst)
print(n)
```

```
4
```

1.2.5 Dictionaries

Dictionaries are container objects that map keys to values.

- Both keys and values can be (almost any) Python objects, even though usually we use strings as keys.
- Dictionaries are created using curly braces: `{key1: value1, key2: value2, ...}`.
- Dictionaries are unordered, ie. elements stored in dictionaries cannot be retrieved in any guaranteed order.

[18]:
```
dct = {'institution': 'University of Glasgow',
       'course': 'Python bootcamp'}
dct
```

[18]:
```
{'institution': 'University of Glasgow', 'course': 'Python bootcamp'}
```

Specific values are accessed using the syntax `dict[key]`:

[19]:
```
value = dct['institution']
value
```

[19]:
```
'University of Glasgow'
```

1.3 NumPy arrays

NumPy is a library that allows us to efficiently store and access (mainly) numerical data and apply numerical operations similar to those available in Matlab.

- NumPy is not part of the core Python project.
- Python itself has an array type, but there is really no reason to use it. Use NumPy!
- NumPy types and functions are not built-in, we must first import them to make them visible. We do this using the `import` statement.

The convention is to make NumPy functionality available using the `np` namespace:

[20]:
```
import numpy as np
```

1.3.1 Creating arrays

NumPy offers a multitude of functions to create arrays.

[21]:
```
# Create a 1-dimensional array with 10 elements, initialise values to 0
# We need to prefix the NumPy function zeros() with 'np'
arr = np.zeros(10)
arr
```

[21]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

[22]:
```
arr1 = np.ones(5) # vector of five ones
arr1
```

[22]:
```
array([1., 1., 1., 1., 1.])
```

We can also create sequences of integers

[23]:
```
arr2 = np.arange(5)      # vector [0,1,2,3,4]
arr2
```

[23]:
```
array([0, 1, 2, 3, 4])
```

`np.arange()` accepts initial values and increments as optional arguments. The end value is NOT included.

[24]:
```
start = 2
end = 10
step = 2
arr3 = np.arange(start, end, step)
arr3
```

[24]:
```
array([2, 4, 6, 8])
```

As in Matlab, there is a `np.linspace()` function that creates a vector of uniformly-spaced real values.

[25]:
```
# Create 11 elements, equally spaced on the interval [0.0, 1.0]
arr5 = np.linspace(0.0, 1.0, 11)
arr5
```

[25]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

We create arrays of higher dimension by specifying the desired shape. Shapes are specified as `tuple` arguments; for example, the shape of an $m \times n$ matrix is `(m,n)`.

[26]:
```
mat = np.ones((2,2))     # Create 2x2 matrix of ones
mat
```

[26]:
```
array([[1., 1.],
       [1., 1.]])
```

Creating arrays from other Python objects Arrays can be created from other objects such as lists and tuples by calling `np.array()`

[27]:
```
# Create array from list [1,2,3]
arr = np.array([1,2,3])
```

```
arr
```

[27]:
```
array([1, 2, 3])
```

[28]:
```
# Create array from tuple
arr = np.array((1.0,2.0,3.0))
arr
```

[28]:
```
array([1., 2., 3.])
```

[29]:
```
# Create two-dimensional array from nested list
arr = np.array([[1,2,3],[4,5,6]])
arr
```

[29]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

1.3.2 Reshaping arrays

The `reshape()` method of an array object can be used to reshape it to some other (conformable) shape.

[30]:
```
mat = np.arange(4).reshape((2,2))
mat
```

[30]:
```
array([[0, 1],
       [2, 3]])
```

[31]:
```
# reshape back to vector of 4 elements
vec = mat.reshape(4)
vec
```

[31]:
```
array([0, 1, 2, 3])
```

We use `-1` to let NumPy automatically compute the size of one remaining dimension.

[32]:
```
# with 2 dimensions, second dimension must have size 2
mat = np.arange(4).reshape((2,-1))
mat
```

[32]:
```
array([[0, 1],
       [2, 3]])
```

If we want to convert an arbitrary array to a vector, we can alternatively use the `flatten()` method.

[33]:
```
mat.flatten()
```

[33]:
```
array([0, 1, 2, 3])
```

Important: the reshaped array must have the same number of elements!

[34]:
```
import numpy as np
mat = np.arange(6).reshape((2,-1))
mat.reshape((2,2))      # Cannot reshape 6 into 4 elements!
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-d98d07ea8aa9> in <module>
      1 import numpy as np
      2 mat = np.arange(6).reshape((2,-1))
----> 3 mat.reshape((2,2))      # Cannot reshape 6 into 4 elements!
```

```
ValueError: cannot reshape array of size 6 into shape (2,2)
```

1.3.3 Indexing

Single element indexing To retrieve a single element, specify the element's index on each axis (axis is the NumPy terminology for an array dimension).

- Remember that just like Python in general, NumPy arrays use 0-based indices.
- Unlike lists or tuples, NumPy arrays support multi-dimensional indexing.

[35]:
```
import numpy as np

mat = np.arange(6).reshape((3,2))
mat
```

[35]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

[36]:
```
mat[1,1]    # returns element in row 2, column 2
```

[36]:
```
3
```

It is important to pass multi-dimensional indices as a tuple to within brackets, ie `[1,1]` in the above example. We could alternatively write `mat[1][1]`, which would give the same result:

[37]:
```
mat[1][1] == mat[1,1]       # don't do this!
```

[37]:
```
True
```

This is substantially less efficient, though, as it first creates a sub-dimensional array `mat[1]`, and then applies the second index to this array.

Index slices There are numerous ways to retrieve a subset of elements from an array. The most common way is to specify a triplet of values `start:stop:step` called `slice` for some axis:

[38]:
```
# Create a 3x2 matrix
mat = np.arange(6).reshape((2,3))
mat
```

[38]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[39]:
```
# Retrieve only the first and third columns:
mat[0:2,0:3:2]
```

[39]:
```
array([[0, 2],
       [3, 5]])
```

Indexing with slices can get quite intricate. Some basic rules:

- all tokens in `start:stop:step` are optional, with the obvious default values. We could therefore write `::` to include all indices, which is the same as `:`
- The end value is NOT included. Writing `vec[0:n]` does not include element with index n!
- Any of the elements of `start:stop:step` can be negative.
  - If `start` or `stop` are negative, elements are counted backwards: `vec[:-1]` retrieves the whole vector except for the last element.
  - If `step` is negative, the order of elements is reversed.

[40]:
```
vec = np.arange(5)
# These are equivalent ways to return the WHOLE vector
vec[0:5:1]      # all three tokens present
vec[::]         # omit all tokens
vec[:]          # omit all tokens
vec[:5]         # end value only
vec[-5:]        # start value only, using negative index
```

[40]:
```
array([0, 1, 2, 3, 4])
```

You can reverse the order like this:

[41]:
```
vec[::-1]
```

[41]:
```
array([4, 3, 2, 1, 0])
```

With multi-dimensional arrays, the above rules apply for each dimension.

- We can, however, omit explicit indices for higher-order dimensions if all elements should be included.

[42]:
```
mat[1]      # includes all columns of row 2; same as mat[1,:]
```

[42]:
```
array([3, 4, 5])
```

We cannot omit the indices for preceding axes, though!

[43]:
```
mat[:, 1]   # includes all rows of column 2
```

[43]:
```
array([1, 4])
```

Indexing lists and tuples The basic indexing rules we have covered so far also apply to the built-in `tuple` and `list` types. However, `list` and `tuple` do not support advanced indexing available for NumPy arrays which we study in later units.

[44]:
```
# Apply start:stop:step indexing to tuple
tpl = (1,2,3)
tpl[:3:2]
```

[44]:
```
(1, 3)
```

1.3.4 Numerical data types (advanced)

You can explicitly specify the numerical data type when creating NumPy arrays.

So far we haven't done so, and then NumPy does the following:

- Functions such as `zeros()` and `ones()` default to using `np.float64`, a 64-bit floating-point data type (this is also called *double precision*)
- Other functions such as `arange()` and `array()` inspect the input data and return a corresponding array.
- Most array creation routines accept a `dtype` argument which allows you to explicitly set the data type.

Examples

[45]:
```
# Floating-point arguments return array of type np.float64
arr = np.arange(1.0, 5.0, 1.0)
arr.dtype
```

[45]:
```
dtype('float64')
```

[46]:
```
# Integer arguments return array of type np.int64
arr = np.arange(1,5,1)
arr.dtype
```

[46]:
```
dtype('int64')
```

Often we don't care about the data type too much, but keep in mind that

- Floating-point has limited precision, even for integers larger than (approximately) 10^{16}
- Integer values cannot represent fractional numbers and (often) have a more limited range.

This might lead to surprising consequences:

[47]:
```
# Create integer array
arr = np.ones(5, dtype=np.int64)
# Add floating-point value 0.234 to second element in place
arr[1] += 0.234
arr
```

[47]:
```
array([1, 1, 1, 1, 1])
```

The array is unchanged because it's impossible to represent 1.234 as an integer value!

The take-away is to always explicitly write floating-point literal values of specify a floating-point `dtype` argument when we want data to be interpreted as floating-point values, eg. always write 1.0 instead of 1, unless you *really* want an integer!

2 Exercises

2.1 Exercise 1: string operations

Experiment with operators applied to strings and integers.

1. Define two string variables and concatenate them using `+`
2. Define a string variable and multiply it by 2 using `*`. What happens?
3. Define two strings and compare whether they are equal using the `==` and `!=` relational operators.
4. Define a string. Use the operators `in` and `not in` to test whether a character is contained in the string.
5. Define two string variables and assign them the same value. Use the `is` operator to test whether these are identical objects.

2.2 Exercise 2: string formatting

We frequently want to create strings that incorporate integer and floating-point data, possibly formatted in a particular way.

Python offers quite powerful formatting capabilities which can become so complex that they are called the *Format Specification Mini-Language* (see the docs). In this exercise, we explore a small but useful subset of formatting instructions.

A format specification is a string that contains one or several `{}`, for example:

```
s = 'The current version of Python is {}'
```

The token `{}` will be replaced with data converted to a string when we apply the `format()` method:

```
s = 'The current version of Python is {}'.format(3.9)
```

The string `s` now contains the value

```
'The current version of Python is 3.9'
```

What if we want to format the float `3.9` in a particular way? We can augment the `{}` to achieve that goal. For example, if the data to be formatted is of type integer, we can specify - `{:wd}` where `w` denotes the total field width and `d` indicates that the data type is an integer.

```
To print an integer into a field that is 3 characters
wide, we would thus write `{:3d}`.
```

For floats we have additional options: - `{:w.df}` specifies that a float should be formatted using a field width `w` and `d` decimal digits.

```
To print a float into a field of 10 characters using 5 decimal
digits, we would thus specify `{:10.5f}`
```

- `{:w.de}` is similar, but instead uses scientific notation with exponents.

  This is particularly useful for very large or very small numbers.

- `{:w.dg}`, where `g` stands for *general* format, is a superset of `f` and `e` formatting. Either fixed or exponential notation is used depending on a numbers magnitude.

In all these cases the field width `w` is optional and can be omitted. Python then uses however many characters are required.

Now what we have introduced the formatting language, you are asked to perform the following exercises:

1. Define two strings and concatenate them using the `format()` function. Add a space between them.
2. Use the above example format string, but truncate the Python version to its major version number. Do you get the expected result?
3. Print π using a precision of 10 decimal digits. *Hint:* the value of π is available as `from math import pi`
4. Print `exp(10.0)` using exponential notation and three decimal digits. *Hint:* To use the exponential function, you need to import it usding `from math import exp`

3 Solutions

3.1 Solution for exercise 1

[48]:
```
# 1. string concatenation using addition
str1 = 'abc'
str2 = 'xyz'

# Concatenate two strings using +
str1 + str2
```

[48]:
```
'abcxyz'
```

[49]:
```
# 2. string multiplication by integers
str1 = 'abc'
# Repeat string using multiplication!
str1 * 2
```

[49]:
```
'abcabc'
```

[50]:
```
# 3. Test for string equality
str1 = 'abc'
str2 = 'xyz'
str1 == str2
```

[50]: False

[51]:
```
# 3. Test for string inequality
str1 = 'abc'
str2 = 'xyz'
str1 != str2
```

[51]: True

[52]:
```
# 4. Test whether individual character is included in string
str1 = 'abc'
'b' in str1
```

[52]: True

[53]:
```
# 4. Test whether individual character is NOT included in string
str1 = 'abc'
'x' not in str1
```

[53]: True

The last two examples illustrate that in Python strings will be interpreted as collections (of characters), just like lists or tuples, if the context requires it. We can therefore apply the `in` operator to test for membership.

[54]:
```
# 5. Test for identity
str1 = 'abc'
str2 = 'abc'
str1 is str2
```

[54]: True

This result should be surprising and is somewhat specific to Python. We would not expect two objects that were created completely independently from each other to be *identical*, ie point to the same memory. Python, however, caches string literals for reasons of efficiency, so it actually does keep only *one* copy of `'abc'` around, irrespective of how many variables containing `abc` are created.

3.2 Solution for exercise 2

[55]:
```
# 1. String concatenation
str1 = 'abc'
str2 = 'xyz'

# format specification to concatenate two string with a space inbetween
fmt = '{} {}'
fmt.format(str1, str2)
```

[55]: 'abc xyz'

[56]:
```
# 2. Truncate Python version to major version number
# To do this, we specify 0 decimal digits!
fmt = 'The current version of Python is {:.0f}'
fmt.format(3.9)      # Now this didn't work out as intended :)
```

[56]: 'The current version of Python is 4'

This does not work as intended because formatting with zero decimal digits rounds the floating-point number, instead of just truncating the decimal part.

[57]:
```
# 3. Print pi using 10 decimal digits
from math import pi
fmt = 'The first 10 digits of pi: {:.10f}'
fmt.format(pi)
```

[57]:
```
'The first 10 digits of pi: 3.1415926536'
```

[58]:
```
# 4. Print exp(10.0) using three decimal digits and exponential notation
from math import exp
fmt = 'exp(10.0) = {:.3e}'
fmt.format(exp(10.0))
```

[58]:
```
'exp(10.0) = 2.203e+04'
```