HandyPython Documentation

0.1

zeping.Long

2018 07 22

:

CHAPTER 1

CHAPTER 2

;JupyterLinux Python Numpy Matplotlib;

2.1 linux

hello world

test markdown

2.2 iPython

2.2.1

- jupyter notebook

- microsfot Azure

2.2.2

1. pip install jupyter

1. pip install sphinx

2.2.3

2.3 numpy

2.4

2.5 An Informal Introduction to Python

[The source material is from Python 3.5.1, but the contents of this tutorial should apply to almost any version of Python 3]

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

2.5.1 Using Python as a Calculator

Let's try some simple Python commands.

Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, −, ⋆ and / work just like in most other languages (for example, Pascal or C); parentheses (( )) can be used for grouping. For example:

```
4
```

```
20
```

```
5.0
```

```
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int <https://docs.python.org/3.5/library/functions.html#int>`__, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float <https://docs.python.org/3.5/library/functions.html#float>`__. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the `//` operator; to calculate the remainder you can use `%`:

```
5.666666666666667
```

```
5
```

```
2
```

```
17
```

With Python, it is possible to use the `**` operator to calculate powers:

```
25
```

```
128
```

Do note that `**` has higher precedence than `-`, so if you want a negative base you will need parentheses:

```
-9
```

```
9
```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
900
```

If a variable is not defined (assigned a value), trying to use it will give you an error:

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-15-3f95c5a59d25> in <module>()
----> 1 n  # Try to access an undefined variable.


NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
7.5
```

```
3.5
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
12.5625
```

```
113.0625
```

```
113.06
```

This variable should be treated as read-only by the user. Don't explicitly assign a value to it, you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal <https://docs.python.org/3.5/library/decimal.html#decimal.Decimal>`__ and `Fraction <https://docs.python.org/3.5/library/fractions.html#fractions.Fraction>`__. Python also has built-in support for complex numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result. `\` can be used to escape quotes:

```
'spam eggs'
```

```
"doesn't"
```

```
"doesn't"
```

```
'"Yes," he said.'
```

```
'"Yes," he said.'
```

```
'"Isn't," she said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print()` <https://docs.python.org/3.5/library/functions.html#print>`__ function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
'"Isn't," she said.'
```

```
"Isn't," she said.
```

```
'First line.nSecond line.'
```

```
First line.
Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
C:some
ame
```

```
C:somename
```

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. The following example:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
'ununununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
'Python'
```

This only works with two literals though, not with variables or expressions:

```
  File "<ipython-input-36-6fcf19fbd400>", line 2
    prefix 'thon'  # Can't concatenate a variable and a string literal.
                 ^
SyntaxError: invalid syntax
```

```
  File "<ipython-input-37-826b8aeb7d3b>", line 1
    ('un' * 3) 'ium'
                 ^
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use +:

```
'Python'
```

This feature is particularly useful when you want to break long strings:

```
'Put several strings within parentheses to have them joined together.'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
'P'
```

```
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
'n'
```

```
'o'
```

```
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

```
'Py'
```

```
'tho'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
'Python'
```

```
'Python'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
'Py'
```

```
'on'
```

```
'on'
```

One way to remember how slices work is to think of the indices as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

P	y	t	h	o	n

0 1 2 3 4 5 6

-6 **-5** **-4** **-3** **-2**

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-52-80def1e3ff80> in <module>()
----> 1 word[42]  # The word only has 6 characters.


IndexError: string index out of range
```

```
'on'
```

```
''
```

Python strings cannot be changed, they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-55-197b67ffdd83> in <module>()
----> 1 word[0] = 'J'


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

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-56-0786521edfdd> in <module>()
----> 1 word[2:] = 'py'


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

```
'Jython'
```

```
'PyPy'
```

The built-in function `len() <https://docs.python.org/3.5/library/functions.html#len>`__ returns the length of a string:

```
34
```

See also:

- Text Sequence Type str: Strings are examples of *sequence types*, and support the common operations supported by such types.
- String Methods: Strings support a large number of methods for basic transformations and searching.
- Format String Syntax: Information about string formatting with `str.format() <https://docs.python.org/3.5/library/string.html#formatstrings>`__.
- `printf-style String Formatting <https://docs.python.org/3.5/library/stdtypes.html#old-string-formatting>`__: The old formatting operations invoked when strings and Unicode strings are the left operand of the % operator.

Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence type), lists can be indexed and sliced:

```
1
```

```
25
```

```
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
[1, 4, 9, 16, 25]
```

Lists also support operations like concatenation:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
64
```

```
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `append()` method (we will see more about methods later):

```
[1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

```
['a', 'b', 'f', 'g']
```

```
[]
```

The built-in function `` `len() <https://docs.python.org/3.5/library/functions.html#len>`__ also applies to lists:

```
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
[['a', 'b', 'c'], [1, 2, 3]]
```

```
['a', 'b', 'c']
```

```
'b'
```

2.5.2 First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The `` `while <https://docs.python.org/3.5/reference/compound_stmts.html#while>`__ `` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), == (equal to), `<`= (less than or equal to), `>`= (greater than or equal to) and `!`= (not equal to).
- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The `` `print() <https://docs.python.org/3.5/library/functions.html#print>`__ `` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
The value of i is 65536
```

```
3
```

```
0.09988002399520096
```

CHAPTER 3

tushare package

3.1 Subpackages

3.1.1 tushare.bond package

Submodules

tushare.bond.bonds module

Module contents

3.1.2 tushare.coins package

Submodules

tushare.coins.market module

Module contents

3.1.3 tushare.data package

Module contents

3.1.4 tushare.fund package

Submodules

tushare.fund.cons module

tushare.fund.nav module

Module contents

3.1.5 tushare.futures package

Submodules

CHAPTER 4

Indices and tables

- genindex
- modindex
- search