**Important**

This documentation covers IPython versions 6.0 and higher. Beginning with version 6.0, IPython stopped supporting compatibility with Python versions lower than 3.3 including all versions of Python 2.7.

If you are looking for an IPython version compatible with Python 2.7, please use the IPython 5.x LTS release and refer to its documentation (LTS is the long term support release).

# Python vs IPython

This document is meant to highlight the main differences between the Python language and what are the specific constructs you can do only in IPython.

Unless expressed otherwise all of the constructs you will see here will raise a `SyntaxError` if run in a pure Python shell, or if executing in a Python script.

Each of these features is described more in detail in the further parts of the documentation.

## Quick overview:

All the following constructs are valid IPython syntax:

```
In [1]: ?
```

```
In [1]: ?object
```

```
In [1]: object?
```

```
In [1]: *pattern*?
In [1]: %shell like --syntax
```

```
In [1]: !ls
```

```
In [1]: my_files = !ls ~/
In [1]: for i, file in enumerate(my_files):
   ...:     raw = !echo $file
   ...:     !echo {file[0].upper()} $raw
```

```
In [1]: %%perl magic --function
   ...: @months = ("July", "August", "September");
   ...: print $months[0];
```

Each of these constructs is compiled by IPython into valid python code and will do most of the time what you expect it will do. Let's see each of these examples in more detail.

## Accessing help

As IPython is mostly an interactive shell, the question mark is a simple shortcut to get help. A question mark alone will bring up the IPython help:

```
In [1]: ?

IPython -- An enhanced Interactive Python
=========================================

IPython offers a combination of convenient shell features, special commands
and a history mechanism for both input (command history) and output (results
caching, similar to Mathematica). It is intended to be a fully compatible
replacement for the standard Python interpreter, while offering vastly
improved functionality and flexibility.

At your system command line, type 'ipython -h' to see the command line
options available. This document only describes interactive features.

MAIN FEATURES
-------------
...
```

A single question mark before or after an object available in the current namespace will show help relative to this object:

```
In [6]: object?
Docstring: The most base type
Type:      type
```

A double question mark will try to pull out more information about the object, and if possible display the python source code of this object.

```
In[1]: import collections
In[2]: collections.Counter??

Init signature: collections.Counter(*args, **kwds)
Source:
class Counter(dict):
    '''Dict subclass for counting hashable items.  Sometimes called a bag
    or multiset.  Elements are stored as dictionary keys and their counts
    are stored as dictionary values.

    >>> c = Counter('abcdeabcdabcaba')  # count elements from a string

    >>> c.most_common(3)                # three most common elements
    [('a', 5), ('b', 4), ('c', 3)]
    >>> sorted(c)                       # list all unique elements
    ['a', 'b', 'c', 'd', 'e']
    >>> ''.join(sorted(c.elements()))   # list elements with repetitions
    'aaaaabbbbcccdde'
    ...
```

If you are looking for an object, the use of wildcards `*` in conjunction with a question mark will allow you to search the current namespace for objects with matching names:

```
In [24]: *int*?
FloatingPointError
int
print
```

## Shell Assignment

When doing interactive computing it is a common need to access the underlying shell. This is doable through the use of the exclamation mark `!` (or bang).

This allows to execute simple commands when present in beginning of the line:

```
In[1]: !pwd
/User/home/
```

Change directory:

```
In[1]: !cd /var/etc
```

Or edit file:

```
In[1]: !mvim myfile.txt
```

The line after the bang can call any program installed in the underlying shell, and support variable expansion in the form of `$variable` or `{variable}`. The later form of expansion supports arbitrary python expressions:

```
In[1]: file = 'myfile.txt'

In[2]: !mv $file {file.upper()}
```

The bang (`!`) can also be present on the right hand side of an assignment, just after the equal sign, or separated from it by a white space. In this case the standard output of the command after the bang will be split out into lines in a list-like object and assigned to the left hand side.

This allows you, for example, to put the list of files of the current working directory in a variable:

```
In[1]: my_files = !ls
```

You can combine the different possibilities in for loops, conditions, functions...:

```
my_files = !ls ~/
for i, file in enumerate(my_files):
    raw = !echo $backup $file
    !cp $file {file.split('.')[0] + '.bak'}
```

## Magics

Magic functions (magics) are often present in the form of shell-like syntax, but they are python functions under the hood. The syntax and assignment possibilities are similar to the one with the bang (`!`) syntax, but with more flexibility and power. Magic functions start with a percent sign (`%`) or double percent signs (`%%`).

A magic call with a single percent sign will act only on one line:

```
In[1]: %xmode
Exception reporting mode: Verbose
```

Magics support assignment:

```
In [1]: results = %timeit -r1 -n1 -o list(range(1000))
62.1 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

In [2]: results
<TimeitResult : 62.1 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)>
```

Magics with double percent signs (`%%`) can spread over multiple lines, but they do not support assignments:

```
In[1]: %%bash
...   : echo "My shell is:" $SHELL
...   : echo "My disk usage is:"
...   : df -h
My shell is: /usr/local/bin/bash
My disk usage is:
Filesystem      Size   Used  Avail Capacity  iused   ifree %iused  Mounted on
/dev/disk1     233Gi  216Gi   16Gi    94% 56788108 4190706   93%   /
devfs          190Ki  190Ki    0Bi   100%      656       0  100%   /dev
map -hosts       0Bi    0Bi    0Bi   100%        0       0  100%   /net
map auto_home    0Bi    0Bi    0Bi   100%        0       0  100%   /hom
```