

My Python Keynotes

```
print("""\
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
""")
```

\ => prevents auto-newline

```
>>> word = 'Python'
>>> word[5] # character in position 5
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

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

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

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

Python strings cannot be changed — they are **immutable**. Therefore, assigning to an indexed position in the string results in an error. If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Lists

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
>>> squares + [36, 49, 64, 81, 100] # concatenation is supported
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

append() can be used (e.g. cubes.append(216))

```
>>> x = [['a', 'b', 'c'], [1, 2, 3]]
```

First Steps Towards Programming

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
```

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.

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).

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Since `**` has higher precedence than `-`, `-3**2` will be interpreted as `-(3**2)` and thus result in `-9`. To avoid this and get 9, you can use `(-3)**2`.

Control Flow Tools

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
>>> for w in words[:]: # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

With `for w in words:`, the example would attempt to create an infinite list, inserting `defenestrate` over and over again.

```
>>> for i in range(5):
...     print(i, end=" ")
```

```
...
0 1 2 3 4
```

```
range(5, 10)
    5, 6, 7, 8, 9

range(0, 10, 3)
    0, 3, 6, 9

range(-10, -100, -30)
    -10, -40, -70
```

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

A strange thing happens if you just print a range:

```
>>> print(range(10))
range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such an iterator. The function `list()` is another; it creates lists from iterables:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

a loop's `else` clause runs when no `break` occurs:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
```

```
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
... 
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
... 
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
```

Defining Functions

```
>>> def fib(n): # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring. There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

Global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

```
ok = input(prompt)
if ok in ('y', 'ye', 'yes'):
    return True
```

...

... To be continued

...