
The Basics

A Code Sample

```
x = 34 - 23          # A comment.
y = "Hello"          # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"  # String concat.
print x
print y
```

Enough to Understand the Code

- Assignment uses `=` and comparison uses `==`.
- For numbers `+` `-` `*` `/` `%` are as expected.
 - Special use of `+` for string concatenation.
 - Special use of `%` for string formatting (as with `printf` in C)
- Logical operators are words (`and`, `or`, `not`)
not symbols
- The basic printing command is `print`.
- The first assignment to a variable creates it.
 - Variable types don't need to be declared.
 - Python figures out the variable types on its own.

Basic Datatypes

- **Integers (default for numbers)**

`z = 5 / 2` `# Answer is 2, integer division.`

- **Floats**

`x = 3.456`

- **Strings**

- Can use `"` or `'` to specify.

`"abc"` `'abc'` (Same thing.)

- Unmatched can occur within the string.

`"matt's"`

- Use triple double-quotes for multi-line strings or strings than contain both `'` and `"` inside of them:

`"""a'b'c"""`

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- **Use a newline to end a line of code.**
 - Use `\` when must go to next line prematurely.
- **No braces `{ }` to mark blocks of code in Python... Use *consistent* indentation instead.**
 - The first line with *less* indentation is outside of the block.
 - The first line with *more* indentation starts a nested block
- **Often a colon appears at the start of a new block. (E.g. for function and class definitions.)**

Comments

- Start comments with # – the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):  
    """This is the docstring. This  
    function does blah blah blah."""  
    # The code would go here...
```

Assignment

- ***Binding a variable*** in Python means setting a *name* to hold a *reference* to some *object*.
 - *Assignment creates references, not copies*
- **Names in Python do not have an intrinsic type. Objects have types.**
 - Python determines the type of the reference automatically based on the data object assigned to it.
- **You create a name the first time it appears on the left side of an assignment expression:**
`x = 3`
- **A reference is deleted via garbage collection after any names bound to it have passed out of scope.**

Accessing Non-Existent Names

- If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#16>", line 1, in -toplevel-
```

```
    y
```

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

```
>>> y = 3
```

```
>>> y
```

```
3
```


Multiple Assignment

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
```

```
>>> x
```

```
2
```

```
>>> y
```

```
3
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

bob Bob _bob _2_bob_ bob_2 BoB

- There are some reserved words:

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

Sequence types:

Tuples, Lists, and Strings

Sequence Types

1. Tuple

- A simple *immutable* ordered sequence of items
- Items can be of mixed types, including collection types

2. Strings

- *Immutable*
- **Conceptually very much like a tuple**

3. List

- *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
 - Tuples and strings are *immutable*
 - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
 - most examples will just show the operation performed on one

Sequence Types 1

- **Tuples are defined using parentheses (and commas).**

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- **Lists are defined using square brackets (and commas).**

```
>>> li = ["abc", 34, 4.34, 23]
```

- **Strings are defined using quotes (" , ' , or "" "").**

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

```
>>> st = """This is a multi-line  
string that uses triple quotes."""
```

Sequence Types 2

- We can access individual members of a tuple, list, or string using square bracket “array” notation.
- *Note that all are 0 based...*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
'abc'
```

```
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
34
```

```
>>> st = "Hello World"
>>> st[1]      # Second character in string.
'e'
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2, 3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]  
'abc'
```

Negative lookup: count from right, starting with -1.

```
>>> t[-3]  
4.56
```


Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]  
( 'abc', 4.56, (2,3) )
```

You can also use negative indices when slicing.

```
>>> t[1:-1]  
( 'abc', 4.56, (2,3) )
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2, 3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]  
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]  
(4.56, (2, 3), 'def')
```

Copying the Whole Sequence

To make a *copy* of an entire sequence, you can use `[:]`.

```
>>> t[:]  
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1      # 2 names refer to 1 ref  
                        # Changing one affects both
```

```
>>> list2 = list1[:]   # Two independent copies, two refs
```

The 'in' Operator

- **Boolean test whether a value is inside a container:**

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- **For strings, tests for substrings**

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- **Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*.**

The + Operator

- The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"
'Hello World'
```

The * Operator

- The * operator produces a *new* tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> "Hello" * 3  
'HelloHelloHello'
```

Mutability: Tuples vs. Lists

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
```

```
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```


Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- We can change lists *in place*.
- Name *li* still points to the same memory reference when we're done.
- The mutability of lists means that they aren't as fast as tuples.

The *extend* method vs the **+** operator.

- **+** creates a fresh list (with a new memory reference)
- *extend* operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Confusing:

- **Extend** takes a list as an argument.
- **Append** takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']
```

```
>>> li.index('b')      # index of first occurrence  
1
```

```
>>> li.count('b')      # number of occurrences  
2
```

```
>>> li.remove('b')     # remove first occurrence  
>>> li  
['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]
```

```
>>> li.reverse()      # reverse the list *in place*
```

```
>>> li  
[8, 6, 2, 5]
```

```
>>> li.sort()         # sort the list *in place*
```

```
>>> li  
[2, 5, 6, 8]
```

```
>>> li.sort(some_function)  
# sort in place using user-defined comparison
```

Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
 - Lists can be modified, and they have lots of handy operations we can perform on them.
 - Tuples are immutable and have fewer features.
- **To convert between tuples and lists use the `list()` and `tuple()` functions:**

```
li = list(tu)
tu = tuple(li)
```