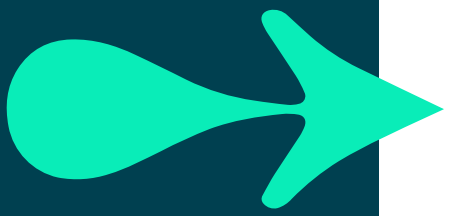
POWERING POTENTIAL

Python 3 Programming

Flow Control

FLOW CONTROL



Contents

- Python conditionals
- What is truth?
- Boolean and logical operators
- Chained comparisons
- Sequence and collection tests
- Object types
- While loops
- For loops
- Conditional expressions
- Unconditional closedown

Now we look at decision making, if statements and loops. Finally, we look at how to give-up and exit our program.

Python conditionals

Conditional membership is by *indentation*

- Designed for readability
- *Syntax:*

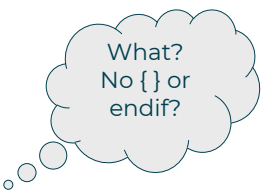
```
if condition:
    statements
elif condition:
    statements
else:
    statements
```

- Boolean operators are overloaded by type

→ **No need for different text or numeric operators**

```
if lista == listb:
    print("Same!")

if "eggs" in lista:
    print("It eggists!")
```



The Python syntax for conditional statements is very simple. The condition is terminated by a colon, and membership of the block which follows is by consistent indentation. The indentation of the first line of the block is expected in lines which follow. A change in indentation, marks the end of the block (or an error if Python cannot figure out what you mean). The white-space used can be any amount, but must be consistent.

The usual set of boolean operators are supported:

> < == >= <= or and not in

No special operators are required for built-in types (like comparing strings), the standard operators are overloaded.

Indentation

> Indentation?! - I will show you how to indent when I indent your skull!
>
> Python? That is for children. A Klingon Warrior uses only machine code, keyed in on the front panel switches in raw binary.
>
> **Klingon Programmer**

> The Future
> We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like *indentation*, rather than delimiters, might become feasible for expressing local structure in the source language.
>
> **Donald Knuth 1974**

Guidelines from PEP008:

- Spaces are the preferred indentation method
- Use 4 spaces per indentation level

Python 3 does not allow mixing tabs and spaces

py3

The Donald Knuth quote is from "Structured Programming with go to Statements", written in 1974.
How much indentation should be used? It is generally thought that 4 spaces is the optimum, but it really does not matter if you disagree - pick the indentation that you feel is the best and stick with that. Tabs are usually a bad idea because their size varies between editors. What looks good in one may be terrible in another.
In addition to the guidelines from PEP008 shown:

Indentation

Use 4 spaces per indentation level.

Tabs or Spaces?

Never mix tabs and spaces

Tabs should be used solely to remain consistent with code that is already indented with tabs.

What is truth?

Built-in function `bool()` tests an object as a Boolean

- False : 0, `None`, empty string, tuple, list, dictionary, set
- True : everything else
- Constants True and False are defined

py3

True and False are not constants in Python 2!

Use double equal signs (==) to compare values

- Overloaded for built-in types
- Use `is` to compare identities of two objects

Sequence types and dictionaries also support `in`

- Tests membership of the container

```
lang = ['Perl', 'Python', 'PHP', 'Ruby']
if 'Python' in lang:
    print('Python is there')
```

py3

`in` was introduced at 2.6

Unlike C based languages, not every statement in Python has a Boolean value. For example, assignments cannot be mis-read as a comparison in Python, unless you specifically wrap it inside a bool. The comparison operator == is overloaded for different classes, and compares the values of two objects. If you wish to test if two variables refer to the same object, use is.
A useful operator (from awk) is in. This tests for membership of any sequence, including strings, lists, and dictionaries (where it tests for a key). For example:

```
slang = "We luv Python"
if 'Python' in slang:
    print ("Python is in slang")

dlang = {'Perl':'sigils',  'Python':'indentation',
'PHP' :'functions', 'Ruby'  :'Rails'}
if 'Python' in dlang:
    print ("Python is there: " + dlang['Python'])
```

Note: Old Python 2 had the dictionary method "has_key", in Python 3 use in.
Talking of Python 2, did you notice that in Python 2 True and False

are variables? So this was legal, if silly:

```
>>> True = False
```

Fortunately, that gives a SyntaxError on Python 3.

Boolean and logical operators

Boolean operators

`<`	**value less than**	*expression* `<` *expression*
`<=`	**value less than or equal**	*expression* `<=` *expression*
`>`	**value greater than**	*expression* `>` *expression*
`>=`	**value greater than or equal**	*expression* `>=` *expression*
`==`	**value equality**	*expression* `==` *expression*
`!=`	**value inequality**	*expression* `!=` *expression*
`is`	**object identity is the same**	*object* `is` *object*

Python 2 also had <> for value inequality

Logical operators

`not`	**logical NOT**	`not` *expression*
`and`	**logical AND**	*expression* `and` *expression*
`or`	**logical OR**	*expression* `or` *expression*

The Boolean (true/false) operators in Python are conventional in most ways. Unlike some languages, however the same operators are overloaded for different types. For example, these operators may be used to test numbers, strings, lists, or dictionaries. What happens when you mix incompatible types is discussed later.
The odd operator out in the list is ‘is’. All the other operators test the values placed either side of them, is tests the identity of the references themselves to see if they refer to the same object.
Python 2 not only had <>, but also a cmp built-in function.

Chained comparisons

- Useful for a testing a range of values

```
if 0 < number < 42 < distance:
    print("number and distance are within range")
else:
    print("number and distance are out of range")
```

- Same as:

```
if 0 < number and number < 42 and 42 < distance:
    print("number and distance are within range")
else:
    print("number and distance are out of range")
```

- Can be combined

```
if 0 < number < 42 and distance != 20:
    ...
```

A useful shortcut for complex tests (in 'if' statements or 'while' loops) are chained comparisons. These enable the programmer to write a more readable test when a range is concerned.
They can become fairly complex, as our second example shows (and can get worse than that). In principle, you should write code that is easy to understand. Only use a chained comparison, if it makes the code simpler - remember the Zen of Python.

Sequence and collection tests

- **An empty string, tuple, list, dictionary, set returns False**

```
mylist = [0, 1, 2, 3]
if mylist:
    print("mylist is True")
```

```
mylist is True
```

- **Sequences also support built-in `all` and `any`**
  - **`all`** returns True if all items in the sequence are true
  - **`any`** returns True if *any* of the items in the sequence are true

```
mylist = [0, 1, 2, 3]

if not all(mylist):
    print("mylist: not all are True")

if any(mylist):
    print("mylist: at least one item is True")
```

```
mylist: not all are True
mylist: at least one item is True
```

An empty list is not the same as a list which has not been defined. A list, and any sequence (like a string), or dictionary, can exist but be empty, in which case it also has the Boolean value False. Sequences also support the all and any built-ins, which are occasionally useful.

Object types

Beware of comparing objects of different types

- Comparison operators may be overloaded
- Do not expect automatic conversion

```
num = 42
txt = '3'

if txt < num:
    print('Wow!')
else:
    print('Doh!')

if int(txt) < num:
    print('Wow!')
else:
    print('Doh!')
```

py3

```
TypeError: unorderable
types: str() < int()
```

Before Python 3 this was not an error, we just got the wrong answer!

```
Wow!
```

A drawback of Python's use of overloaded operators is when a binary operator has two dissimilar objects on each side. In old releases of Python, it guessed which object's operator to use, and its guess was not always what the programmer intended. Fortunately, that has been fixed in Python 3, and we get an error message (Exception) as expected.
It is important to force conversion to the correct type, often using the int or str functions. This is the price we have to pay for simpler syntax, the alternative would be to have multiple operators (Perl) or a plethora of built-in functions for comparisons (PHP).

A note on exception handling

- **An Exception is Python's way of telling you something**

→ Unless handled, it will halt the program

- **Many Python built-in functions can raise an exception**

→ When they wish to indicate some condition

- **Exceptions do not necessarily indicate failure**

→ For example:

  - Search for something which does not exist
  - Unable to open a file

- **At this point in the course, we will just live with them**
- **Later, we will discuss how to handle exceptions**

The TypeError shown on the previous slide was actually an Exception. It can be trapped, and the error message changed, as follows:

```
try:
    if txt < num:
        print ('Wow!')
    else:
        print ('Doh!')
except TypeError:
    print("Invalid types compared", file=sys.stderr)
```

The try block contains code to be tested. Should any of that code (which is often a function call) raise an exception, then it can be trapped, and code executed in the except block to handle it. If an exception list is specified then the handler code will only be executed if the exception is in the list, otherwise the stack will be unwound until a handler is found. Unhandled exceptions terminate the program. Python exception handling also supports finally and else.
We discuss exception handling in more detail later.

While loops

Loop while a condition is true

- Python only supports entry condition loops
- There is no *do…while* loop

With all conditionals, membership is by *indentation*

`while` *condition*`:`
    loop body

```
line = None

while line != 'done':
    line = input('Type "done" to complete: ')
    print('<', line, '>')
```

```
myl = [23, 67, 32, 9, 77]

while myl:
    print(myl.pop() * 2)
```

```
154
18
64
134
46
```

`pop()` on a list removes and returns the last item

Loops follow a similar syntax to the if statements, membership of the body of the loop is by consistent indentation. The usual while loops and list processing for statement are available.
Somewhat unusually, in Python there is no do.. while loop, and no case statement. It is possible to emulate some of the features of a case statement using a dictionary.
The first example is fairly simple, but indicates that the tested variable must already exist. Missing out that initialisation would result in a NameError exception.
The second example loops while the list is true: that it is not empty. We are using the pop() list method (which we shall see later) which removes the last (right-most) item from a list and returns it.

Loop control statements

- **Loop control statements**
  - → `continue` perform next iteration
  - → `break` exit the loop at once
  - → `pass` Empty placeholder (do nothing)
- **The else: clause**
  - → Indicates code to be executed when the while condition is false, or when the for list expires
  - → Including when the loop condition is false on entry

```
i = 1
j = 120
while i < 42:
    i = i * 2
    if i > j: break
else:
    print("Loop expired: ", i)
print("Final value: ", i)
```

The `else` clause is not executed if the loop exits using a `break`

```
Loop expired:  64
Final value:  64
```

The loop control statements continue and break may be familiar from other languages such as C, C++, C#, Java, PHP, awk, ksh, Bash. Their use is often frowned upon, and it is true that redesigning conditionals can often make them unnecessary. However, this in itself can make code more difficult to follow, so our guideline is to use them when they make the code easier to follow.

The "do nothing" pass statement is unusual in languages (COBOL has NEXT SENTENCE) and is a consequence of Python's use of white-space to denote membership. It's use is not confined to loops - pass may be used in if statements, and usefully in an exception handler where we just want to ignore the exception.

The else: clause in a loop is also unusual. Statements that are part of this clause are executed when the loop exits, except for a break. In the example, the break is not executed because i will never exceed j, but if j was lower than 42, for example 12, then the break would be executed and the else: clause would not.

For loops

- **Iterate through a sequence**

→ Often a list or tuple

→ Loop variable holds a copy of each element in turn

- **As with conditionals, membership is by *indentation***

```
for variable in object:
    loop body
```

```
import sys
for arg in sys.argv:
   print("Cmd line argument:", arg)
```

`sys.argv` is a list of the command-line arguments

```
C:\Python>for.py Monday Tuesday Wednesday
Cmd line argument: C:\Python\for.py
Cmd line argument: Monday
Cmd line argument: Tuesday
Cmd line argument: Wednesday
```

Accessing a list or tuple sequentially (iterating) is a common enough requirement, and many languages have a for construct for this purpose. A difference with Python is that the iteration might not be done using a simple integer count, the class may have its own iterator (implemented in the class through __iter__). The for loop is preferable to using your own iterator and counting each element yourself - the class is much better at that sort of thing. The loop variable (arg in the example) holds a copy of each element, so altering the variable value will not alter the list (see later for a solution).

enumerate

Use in loops over any sequence

- Returns a two-item tuple which contains a count and the item at that position in the sequence

```
for idx, arg in enumerate(sys.argv):
    print('index:', idx, 'argument:', arg)
```

... or other object type which supports iteration

- For example, `open` will open a file and return an iterator
- enumerate also takes an optional *start* parameter

```
for nr, line in enumerate(open('brian.txt'), start=1):
    print(nr, line, end="")
```

line numbers start from 1, sequences start at 0

```
1 Some things in life are bad
2 They can really make you mad
3 Other things just make you swear and curse.
```

The enumerate function (added at Python 2.3) enables us to obtain the current position in a sequence, as well as the data item. This function can be used on any sequence: a list, tuple, string, or bytearray; or any object that supports iteration.
Two items are returned from enumerate, the sequential number (starting from zero) and the data item at that position. We can also specify a different start number (introduced at Python 2.6).
On the slide, we show a less obvious use of enumerate, from a file open - we shall be describing file IO in more detail later.

Counting 'for' loops

- **Can use the `range()` builtin**

py3 `range` used to be called `xrange`

`range`([*start*], *stop*[, *step*])

```
for i in range(0, len(some_list)):
    if some_list[i] > 42: some_list[i] += 1
```

- But this maintains its own iterator

```
for i in range(0, len(some_list)):
    print(some_list[i])
```

- Use a system generated one instead

```
for num in some_list:
    print(num)
```

- **But an index is needed to alter the sequence…**

```
for idx, num in enumerate(some_list):
    if num > 42: some_list[idx] += 1
```

The built-in range() (called xrange() in previous Python versions) is often used to produce a list of values for counting. All parameters must be integers, but they can be positive or negative. Notice that the value of the stop parameter is never reached.
Counting loops are popular in most primitive languages for iterating through lists, but often they are not required in Python - it is easier and faster to use a system generated iterator than to maintain your own.
We mentioned earlier that the loop variable only holds a copy of the item in a for loop, so altering it will not alter the sequence. We need an index to be able to do that, and enumerate comes to the rescue. To be fair, enumerate cannot do everything - it does not have stop or step parameters.

Zipping through multiple lists

The zip built-in returns an iterator of tuples

- Wrap `list()` around it to get a list
- Can consume a lot of memory
- Useful for stepping through parallel lists

py3

`zip` used to return a *list* of tuples

```
farms   = ['Home Farm', 'Muckworthy',
           'Scales End', 'Brown Rigg']
squirls = [42, 12, 2, 0]
rabbits = [395, 68, 57, 32]
moles   = [12, 8, 0, 29]

for f, s, r, m in zip(farms, squirls, rabbits, moles):
    print ('Total for', f, ':', s + r + m)
```

*A squirl is a truncated squirrel

```
Total for Home Farm : 449
Total for Muckworthy : 88
Total for Scales End : 59
Total for Brown Rigg : 61
```

The zip builtin function returns an iterator of tuples, and the most common use is shown. When used with a for loop and a tuple of loop variables, each loop variable is set to an item from the corresponding tuple (or list). It avoids having to create our own iterator, as in the classic C-style for loop.

The function is useful in other scenarios. For example, here is a quick way of constructing a dictionary from lists of keys and values:

```
keys = ['Australia','Eire','France','Finland',
        'UK','US']
vals = ['Canberra','Dublin','Paris','Helsinki',
        'London','Washington']
mydict = dict(zip(keys,vals))
```

In Python 2 zip returned a list of tuples, in Python 3 zip returns an iterator of tuples, so Python 2 zlist = zip(keys,vals) becomes

```
zlist = list(zip(keys,vals))
```

in Python 3.

Conditional expressions

- Shorthand for conditionals
  expr1 **`if`** *boolean* **`else`** *expr2*

```
i = 42
j = 3

print("i gt j") if i > j else print ("i lt j")

print("i gt j" if i > j else "i lt j")
```

```
if i > j:
    print("i gt j")
else:
    print("i lt j")
```

These 'if' statements all do the same thing

- No : and elif not allowed

```
-1 if a < b else (+1 if a > b else 0)
```

- Beware of precedence

```
a = 54
answer = a + 5 if a < 42 else 0

answer = a + (5 if a < 42 else 0)
```

Conditional expressions replace the ternary conditional (:?) in C-style languages.
The extra parentheses are required in the last example because otherwise if would add 5 to a only if a < 42, otherwise it would set answer to zero.
You may think that conditional expressions contravene Python's clean style, however they are useful for certain more advanced statements, such as lambda functions.

Unconditional closedown

`os._exit(`*`integer_expression`*`)`

- Cannot be trapped
- Returns *integer_expression* to the caller (usually the shell)

`os.abort()`

- Raises a SIGABRT signal (trappable on UNIX)
- Causes a core dump on UNIX, an exit 3 on Windows

`sys.exit(`*`expression`*`)`

- Raises a `SystemExit` exception which can be trapped
- Returns expression to the caller (usually the shell) if it is an integer
- Prints to stderr if any other type of object
  - Returns 1 to the caller

```
sys.exit("Goodbye")
```

The sys.exit method will shutdown the current process, even if called from a function (covered later). The argument to exit is returned to the caller, which is often a shell program, but could be another application. Many environments (e.g. UNIX) only support a single byte for the return code, so do not return numbers outside the range 0 - 255. For portability reasons, it is wise to stay within that range, even on Windows. By convention, zero is success. Some standard exit codes are defined in the UNIX version of the os module (prefixed os.EX_), but these are not universally adopted.
The main advantage of sys.exit over the more primitive os._exit is that it can be trapped by exception handling. It is recommended that os._exit is only used in special circumstances, such as immediately after a fork (UNIX specific way of creating a new process).
There appears to be a built-in called exit which has the same behaviour as sys.exit, but appearances can be deceptive. In fact, exit is not a built-in function but a site.Quitter callable object which raises a SystemExit exception. In early versions of Python (prior to 2.4), the difference was significant, but not anymore. The site module is usually automatically loaded on start-up, but can be

supressed with the -S command-line option, in which case exit will not work.
If exit or sys.exit are not used, then the Python program will return zero.
Python has an atexit module which enables one or more user-written functions to be run on exit to the program (early versions of Python also had exitfunc, which is now deprecated in favour of atexit). These are not run by os._exit.

Unconditional flow control (2)

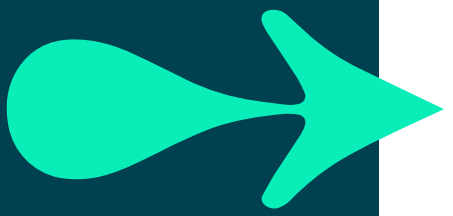
- **"*But I use `exit()` or `quit()`!*"**
- **At start-up, the `site` module is automatically loaded**
  - Unless the -S command-line option is given
  - Several objects are created, including **`exit`** and **`quit`**
- **When *printed*, `exit` and `quit` output a message:**

```
>>> exit
Use exit() or Ctrl-Z plus Return to exit
>>> quit
Use quit() or Ctrl-Z plus Return to exit
```

- **When *called* they raise a `SystemExit` exception and close `stdin`**
  - IDLE ignores `SystemExit` but closes when `stdin` is closed
- **Only use in an interpreter session, not in production code**
  - Because of the side-effect of closing `stdin`

exit and quit are actually objects of site.Quitter class, and the magic behaviour is obtained by the class implementing the special methods __call__ and __repr__ (see later).
Other objects loaded by the site module include copyright, license, and credits, these are objects of class site._Printer.
Quote from the python documentation "They are useful for the interactive interpreter shell and should not be used in programs."
Quite apart from the fact that the site module might not be loaded, the side effect of closing standard input could upset programs that trap the SystemExit exception.

SUMMARY



- **Python has the usual Boolean and logical operators**
- Be careful of types
- **Basic flow control statements :**

if *condition*:
    indented statements

while *condition*:
    indented statements

for *target* in *object*:
    indented statements

- **Terminate a process using `sys.exit()`**