**Course: Advanced Bioinformatics**

**Module title: Python Basics**

**Module no. : 200**

**Syntax and Semantic of Python**

Python uses whitespace to delimit program blocks, following the off-side rule. Python borrows this feature from its predecessor ABC: instead of punctuation or keywords, it uses indentation to indicate the run of a block.

In so-called "free-format" languages, that use the block structure derived from ALGOL, blocks of code are set off with braces ({ }) or keywords. In most coding conventions for these languages, programmers conventionally indent the code within a block, to visually set it apart from the surrounding code (prettyprinting).

Consider a function, foo, which is passed a single parameter, x, and if the parameter is 0 will call bar and baz, otherwise it will call qux, passing x, and also call itself recursively, passing x-1 as the parameter. Here are implementations of this function in both C and Python:

foo function in C with K&R indent style:

void foo(int x)

{

**if** (x == 0) {

bar();

baz();

} **else** {

qux(x);

foo(x - 1);

}

}

foo function in Python:

**def** foo(x):

**if** x == 0:

bar()

baz()

**else**:

qux(x)

foo(x - 1)

**Literals**

**Strings:** Python has various kinds of string literals.

**Normal string literals**

Either single or double quotes can be used to quote strings. Unlike in Unix shell languages, Perl or Perl-influenced languages such as Ruby or Groovy, single quotes and double quotes function identically, i.e. there is no string interpolation of $foo expressions. However, interpolation can be done using the % string-format operator, e.g. the Perl statement

**print** "I just printed $num pages to the printer $printer\n"

is equivalent to the Python statement

**print**("I just printed **%s** pages to the printer **%s**" % (num, printer))

However, the current standard for this sort of string formatting is to use the *format* method of strings:

**print**("I just printed {0} pages to the printer {1}".format(num, printer))

**Multi-line string literals**

There are also multi-line strings, which begin and end with a series of three single or double quotes and function like [here documents](https://en.wikipedia.org/wiki/Here_document) in [Perl](https://en.wikipedia.org/wiki/Perl) and [Ruby](https://en.wikipedia.org/wiki/Ruby_(programming_language)).

A simple example with [variable interpolation](https://en.wikipedia.org/wiki/Variable_interpolation) (using the *%* string-format operator) is:

**print**("""Dear **%(recipient)s**,

I wish you to leave Sunnydale and never return.

Not Quite Love,

**%(sender)s**

""" % {'sender': 'Buffy the Vampire Slayer', 'recipient': 'Spike'})

#### Raw strings

Finally, all of the previously mentioned string types come in "[raw](https://en.wikipedia.org/wiki/Raw_string)" varieties (denoted by placing a literal *r* before the opening quote), which do no backslash-interpolation and hence are very useful for [regular expressions](https://en.wikipedia.org/wiki/Regular_expression); compare ["@-quoting"](https://en.wikipedia.org/wiki/C_Sharp_syntax#Literals) in [C#](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)). Raw strings were originally included specifically for regular expressions. Due to limitations of the tokenizer, raw strings may not have a trailing backslash.[[5]](https://en.wikipedia.org/wiki/Python_syntax_and_semantics#cite_note-7) Creating a raw string holding a [Windows](https://en.wikipedia.org/wiki/Windows) path ending with a backslash requires some variety of workaround (commonly, using forward slashes instead of backslashes, since Windows accepts both).

Examples include:

*# A Windows path, even raw strings cannot end in a backslash*

r"C:\Foo\Bar\Baz**\"** # raises SyntaxError

r"C:\Foo\Bar\Baz\ ".rstrip() *# avoids the error by adding and removing trailing space*

*# A regular expression matching a quoted string with possible backslash quoting*

r'"([^"**\\**]|**\\**.)\*"'

*# Reverse the arguments in a two-arg function call, e.g. foo(2, bar) -> foo(bar, 2).*

re.sub(r'\(([^,]\*?),([^,]\*?)\)', r'(\2, \1)', code)

*# Note that this won't work if either argument has parens or commas in it.*

#### Concatenation of adjacent string literals

String literals (using possibly different quote conventions) appearing contiguously and only separated by whitespace (including new lines), are allowed and are aggregated into a single longer string. Thus

title = "One Good Turn: " \

'A Natural History of the Screwdriver and the Screw'

is equivalent to

title = "One Good Turn: A Natural History of the Screwdriver and the Screw"

### Numbers

Numeric literals in Python are of the normal sort, e.g. 0, -1, 3.4, 3.5e-8.

Python has arbitrary-length integers and automatically increases the storage size as necessary. Prior to Python version 3, there were two kinds of integral numbers: traditional fixed size integers and "long" integers of arbitrary range. The conversion to "long" integers was performed automatically when required, and thus the programmer usually didn't have to be aware of the two integral types. In newer language versions the fixed-size integers are completely gone.

Python supports normal [floating point](https://en.wikipedia.org/wiki/IEEE_754-2008) numbers, which are created when a dot is used in a literal (e.g. 1.1), when an integer and a floating point number are used in an expression, or as a result of some mathematical operations ("true division" via the / operator, or exponentiation with a negative exponent).

Python also supports [complex numbers](https://en.wikipedia.org/wiki/Complex_number) natively. Complex numbers are indicated with the J or j suffix, e.g. 3 + 4j.

### Lists, tuples, sets, dictionaries

Python has syntactic support for the creation of container types.

Lists (class list) are mutable sequences of items of arbitrary types, and can be created either with the special syntax

a\_list = [1, 2, 3, "a dog"]

or using normal object creation

a\_second\_list = list()

a\_second\_list.append(4)

a\_second\_list.append(5)

**Tuples** (class tuple) are immutable sequences of items of arbitrary types. There is also a special syntax to create tuples

a\_tuple = 1, 2, 3, "four"

Although tuples are created by separating items with commas, the whole construct is usually wrapped in parentheses to increase readability. An empty tuple is denoted by ().

**Sets** (class set) are mutable containers of items of arbitrary types. The items are not ordered, but sets support iteration over the items. A syntax for set creation appeared in Python 2.7/3.0

some\_set = {0, (), False}

In earlier Python versions, sets would be created by calling initializing the set class with a list argument. Python sets are very much like [mathematical sets](https://en.wikipedia.org/wiki/Set_(mathematics)), and support operations like set [intersection](https://en.wikipedia.org/wiki/Set_(mathematics)#Intersections) and [union](https://en.wikipedia.org/wiki/Set_(mathematics)#Unions).

Python also features a frozenset class for immutable sets.

**Dictionaries** (class dict) are mutable mappings tying keys and corresponding values. Python has special syntax to create dictionaries ({key: value})

a\_dictionary = {"key 1":"value 1", 2:3, 4:[]}

The dictionary syntax is similar to the set syntax, the difference is the presence of colons. The empty literal {} results in an empty dictionary rather than an empty set, which is instead created using the non-literal constructor: set().

## Operators

### Arithmetic

Python includes the +, -, \*, /, % ([modulus](https://en.wikipedia.org/wiki/Modulo_operator)), and \*\* ([exponentiation](https://en.wikipedia.org/wiki/Exponentiation)) operators, with their usual [mathematical precedence](https://en.wikipedia.org/wiki/Order_of_operations).

Traditionally, x / y performed [integer division](https://en.wikipedia.org/wiki/Integer_division) if both x and y were integers (returning the floor of the quotient), and returned a float if either was a float. However, because Python is a dynamically typed language, it was not always possible to tell which operation was being performed, which often led to subtle bugs. For example, with

**def** mean(seq):

**return** sum(seq) / len(seq)

A call to mean([3.0, 4.0]) would return 3.5, but mean([3, 4]) would return 3. If this was not the intended behavior, it was necessary to use a workaround such as

**def** mean(seq):

**return** float(sum(seq)) / len(seq)

To avoid this issue, [a proposal](https://www.python.org/dev/peps/pep-0238/) was made to change the behavior of the Python division operator. In Python 2.2, a new operator // was introduced for floor division, both for integer and floating-point arguments. The / operator was changed so that the quotient of two integers returned a float, but for backwards compatibility, this behavior had to be explicitly requested until Python 3.0.

### Comparison operators

The basic comparison operators such as ==, <, >=, and so forth are used on all manner of values. Numbers, strings, sequences, and mappings can all be compared. Although disparate types (such as a str and an int) are defined to have a consistent relative ordering, this is considered a historical design quirk and will no longer be allowed in Python 3.0.

Chained comparison expressions such as a < b < c have roughly the meaning that they have in mathematics, rather than the unusual meaning found in [C](https://en.wikipedia.org/wiki/C_(programming_language)) and similar languages. The terms are evaluated and compared in order. The operation has [short-circuit semantics](https://en.wikipedia.org/wiki/Minimal_evaluation), meaning that evaluation is guaranteed to stop as soon as a verdict is clear: if a < b is false, c is never evaluated as the expression cannot possibly be true anymore.

For expressions without side effects, a < b < c is equivalent to a < b and b < c. However, there is a substantial difference when the expressions have side effects. a < f(x) < b will evaluate f(x) exactly once, whereas a < f(x) and f(x) < b will evaluate it twice if the value of a is less than f(x) and once otherwise.

### Logical operators

Python 2.2 and earlier does not have an explicit boolean type. In all versions of Python, boolean operators treat zero values or empty values such as "", 0, None, 0.0, [], and {} as false, while in general treating non-empty, non-zero values as true. In Python 2.2.1 the boolean constants True and False were added to the language (subclassed from 1 and 0). The binary comparison operators such as == and > return either True or False.

The boolean operators and and or use [minimal evaluation](https://en.wikipedia.org/wiki/Minimal_evaluation). For example, y == 0 or x/y > 100 will never raise a divide-by-zero exception. Note that these operators return the value of the last operand evaluated, rather than True or False. Thus the expression (4 and 5) evaluates to 5, and (4 or 5) evaluates to 4.

## Functional programming

As mentioned above, another strength of Python is the availability of a [functional programming](https://en.wikipedia.org/wiki/Functional_programming) style. As may be expected, this makes working with lists and other collections much more straightforward.

### List comprehensions[[edit](https://en.wikipedia.org/w/index.php?title=Python_syntax_and_semantics&action=edit&section=21" \o "Edit section: List comprehensions)]

*Main article:*[*List comprehension*](https://en.wikipedia.org/wiki/List_comprehension)

One such construction is the [list comprehension](https://en.wikipedia.org/wiki/List_comprehension), which can be expressed with the following format:

L = [mapping-expression **for** element **in** source-list **if** filter-expression]

Using list comprehension to calculate the first five powers of two:

powers\_of\_two = [2\*\*n **for** n **in** range(1, 6)]

The [Quicksort](https://en.wikipedia.org/wiki/Quicksort) algorithm can be expressed elegantly (albeit inefficiently) using list comprehensions:

**def** qsort(L):

**if** L == []:

**return** []

pivot = L[0]

**return** (qsort([x **for** x **in** L[1:] **if** x < pivot]) +

[pivot] +

qsort([x **for** x **in** L[1:] **if** x >= pivot]))

For further details visit the wiki page of python at https://en.wikipedia.org/wiki/Python\_syntax\_and\_semantics#Indentation.