IntroPython

January 9, 2016

0.1 A short introduction to Python

In this document, we will introduce the main concepts of scripting with **Python**, and see how typical features of a programming language are implemented in this language.

Scripting languages are designed primarily for interpreted execution rather than compiled execution, and to support quick automation of tedious system administration tasks. They provide features focused on ease of programming, trading off speed of execution.

A clear example of this is given by our very first program, the classic "hello, world!":

```
In [2]: print "Hello, World!"
```

```
Hello, World!
```

As you can see, our "program" (more properly, our script) is a very short one. Since we are merely printing out a string of text, the language does not require us to declare a function, use external libraries for printing to the terminal, etc. Everything is resolved with a single line of code.

In our second example, we will look at declaring and using functions. We will therefore write a function that receives a string, and prints it to the terminal, bracketed in a pair of "<" and ">" characters

```
In [3]: def fprint(s):
            print '<', s, '>'
```

We will now use this function, as part of our script.

```
In [12]: fprint("Hello, World")
```

```
< Hello, World >
```

0.2 Loops, conditionals, and variables

Let us have a brief tour of the main features of a language, in terms of variable declaration and use, and of control statements. To do this, let us consider a simple script that prints the prime numbers smaller than a value nchosen by the user.

```
In [32]: n=None
         while not n :
             n=int(raw_input("Type a number: "))
         result = []
         if n<=1 :
                 print "Only 1..."
         else :
                 for i in range(2,n+1):
                     divisible_by = [ j for j in result if i%j==0 ]
                     if len(divisible_by)==0 :
                             result.append(i)
         print result
```

```
Type a number: 10
[2, 3, 5, 7]
```

We have introduced a number of interesting features!

First, variables do not need to be declared before being used – just be careful not to use the value of a variable if you have not created it first by assigning it a value.

None is a special value, which means no value is assigned to the variable – note that None is false in tests!

"Arrays" are handled in a simple but powerful way in Python. Actually, what we are using here are lists, which can be dynamically extended (see the "append" function in the example), but have a similar syntax has C arrays – so we can say the following:

```
In [33]: print result[1]
         print result[-2]
```

```
3
5
```

Control constructs include while, for and if. Note that a : is used in most cases to mark the beginning of the loop or then body, and indentation is used to denote it. The for construct iterates over lists or other sequences. In our example, range builds a list of numbers:

```
In [34]: print range(2,n+1)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

An important and powerful construct is the list comprehension employed in the above example. List comprehensions provide a syntax similar to mathematical expressions used to define sets, making them a compact way to define sequences of values:

```
In [36]: l = [ i*i for i in result if i>2 and i<7 ]
         print l
```

```
[9, 25]
```

With a single line, we have built the ordered set of the values in results (i.e., the primes smaller than n), greater than 2 and smaller than 7, squared.

0.3 Online help and a behind the scenes look at strings

Since scripting languages are oriented towards interactive execution, Python includes an online manual in all versions of its interpreter interface. Let us see what the help function can tell us regarding the string type, str

```
In [14]: help(str)
```

```
Help on class str in module __builtin__:

class str(basestring)
 |  str(object='') -> string
 |
 |  Return a nice string representation of the object.
 |  If the argument is a string, the return value is the same object.
 |
 |  Method resolution order:
 |      str
 |      basestring
```

```
 |      object
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
 |  __format__(...)
 |      S.__format__(format_spec) -> string
 |
 |      Return a formatted version of S as described by format_spec.
 |
 |  __ge__(...)
 |      x.__ge__(y) <==> x>=y
 |
 |  __getattribute__(...)
 |      x.__getattribute__('name') <==> x.name
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __getnewargs__(...)
 |
 |  __getslice__(...)
 |      x.__getslice__(i, j) <==> x[i:j]
 |
 |      Use of negative indices is not supported.
 |
 |  __gt__(...)
 |      x.__gt__(y) <==> x>y
 |
 |  __hash__(...)
 |      x.__hash__() <==> hash(x)
 |
 |  __le__(...)
 |      x.__le__(y) <==> x<=y
 |
 |  __len__(...)
 |      x.__len__() <==> len(x)
 |
 |  __lt__(...)
 |      x.__lt__(y) <==> x<y
 |
 |  __mod__(...)
 |      x.__mod__(y) <==> x%y
 |
 |  __mul__(...)
 |      x.__mul__(n) <==> x*n
```

```
 |
 |  __ne__(...)
 |      x.__ne__(y) <==> x!=y
 |
 |  __repr__(...)
 |      x.__repr__() <==> repr(x)
 |
 |  __rmod__(...)
 |      x.__rmod__(y) <==> y%x
 |
 |  __rmul__(...)
 |      x.__rmul__(n) <==> n*x
 |
 |  __sizeof__(...)
 |      S.__sizeof__() -> size of S in memory, in bytes
 |
 |  __str__(...)
 |      x.__str__() <==> str(x)
 |
 |  capitalize(...)
 |      S.capitalize() -> string
 |
 |      Return a copy of the string S with only its first character
 |      capitalized.
 |
 |  center(...)
 |      S.center(width[, fillchar]) -> string
 |
 |      Return S centered in a string of length width. Padding is
 |      done using the specified fill character (default is a space)
 |
 |  count(...)
 |      S.count(sub[, start[, end]]) -> int
 |
 |      Return the number of non-overlapping occurrences of substring sub in
 |      string S[start:end].  Optional arguments start and end are interpreted
 |      as in slice notation.
 |
 |  decode(...)
 |      S.decode([encoding[,errors]]) -> object
 |
 |      Decodes S using the codec registered for encoding. encoding defaults
 |      to the default encoding. errors may be given to set a different error
 |      handling scheme. Default is 'strict' meaning that encoding errors raise
 |      a UnicodeDecodeError. Other possible values are 'ignore' and 'replace'
 |      as well as any other name registered with codecs.register_error that is
 |      able to handle UnicodeDecodeErrors.
 |
 |  encode(...)
 |      S.encode([encoding[,errors]]) -> object
 |
 |      Encodes S using the codec registered for encoding. encoding defaults
 |      to the default encoding. errors may be given to set a different error
 |      handling scheme. Default is 'strict' meaning that encoding errors raise
```

```
 |      a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
 |      'xmlcharrefreplace' as well as any other name registered with
 |      codecs.register_error that is able to handle UnicodeEncodeErrors.
 |
 |  endswith(...)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      suffix can also be a tuple of strings to try.
 |
 |  expandtabs(...)
 |      S.expandtabs([tabsize]) -> string
 |
 |      Return a copy of S where all tab characters are expanded using spaces.
 |      If tabsize is not given, a tab size of 8 characters is assumed.
 |
 |  find(...)
 |      S.find(sub [,start [,end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  format(...)
 |      S.format(*args, **kwargs) -> string
 |
 |      Return a formatted version of S, using substitutions from args and kwargs.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  index(...)
 |      S.index(sub [,start [,end]]) -> int
 |
 |      Like S.find() but raise ValueError when the substring is not found.
 |
 |  isalnum(...)
 |      S.isalnum() -> bool
 |
 |      Return True if all characters in S are alphanumeric
 |      and there is at least one character in S, False otherwise.
 |
 |  isalpha(...)
 |      S.isalpha() -> bool
 |
 |      Return True if all characters in S are alphabetic
 |      and there is at least one character in S, False otherwise.
 |
 |  isdigit(...)
 |      S.isdigit() -> bool
 |
 |      Return True if all characters in S are digits
```

```
 |      and there is at least one character in S, False otherwise.
 |
 |  islower(...)
 |      S.islower() -> bool
 |
 |      Return True if all cased characters in S are lowercase and there is
 |      at least one cased character in S, False otherwise.
 |
 |  isspace(...)
 |      S.isspace() -> bool
 |
 |      Return True if all characters in S are whitespace
 |      and there is at least one character in S, False otherwise.
 |
 |  istitle(...)
 |      S.istitle() -> bool
 |
 |      Return True if S is a titlecased string and there is at least one
 |      character in S, i.e. uppercase characters may only follow uncased
 |      characters and lowercase characters only cased ones. Return False
 |      otherwise.
 |
 |  isupper(...)
 |      S.isupper() -> bool
 |
 |      Return True if all cased characters in S are uppercase and there is
 |      at least one cased character in S, False otherwise.
 |
 |  join(...)
 |      S.join(iterable) -> string
 |
 |      Return a string which is the concatenation of the strings in the
 |      iterable.  The separator between elements is S.
 |
 |  ljust(...)
 |      S.ljust(width[, fillchar]) -> string
 |
 |      Return S left-justified in a string of length width. Padding is
 |      done using the specified fill character (default is a space).
 |
 |  lower(...)
 |      S.lower() -> string
 |
 |      Return a copy of the string S converted to lowercase.
 |
 |  lstrip(...)
 |      S.lstrip([chars]) -> string or unicode
 |
 |      Return a copy of the string S with leading whitespace removed.
 |      If chars is given and not None, remove characters in chars instead.
 |      If chars is unicode, S will be converted to unicode before stripping
 |
 |  partition(...)
 |      S.partition(sep) -> (head, sep, tail)
```

```
 |
 |      Search for the separator sep in S, and return the part before it,
 |      the separator itself, and the part after it.  If the separator is not
 |      found, return S and two empty strings.
 |
 |  replace(...)
 |      S.replace(old, new[, count]) -> string
 |
 |      Return a copy of string S with all occurrences of substring
 |      old replaced by new.  If the optional argument count is
 |      given, only the first count occurrences are replaced.
 |
 |  rfind(...)
 |      S.rfind(sub [,start [,end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  rindex(...)
 |      S.rindex(sub [,start [,end]]) -> int
 |
 |      Like S.rfind() but raise ValueError when the substring is not found.
 |
 |  rjust(...)
 |      S.rjust(width[, fillchar]) -> string
 |
 |      Return S right-justified in a string of length width. Padding is
 |      done using the specified fill character (default is a space)
 |
 |  rpartition(...)
 |      S.rpartition(sep) -> (head, sep, tail)
 |
 |      Search for the separator sep in S, starting at the end of S, and return
 |      the part before it, the separator itself, and the part after it.  If the
 |      separator is not found, return two empty strings and S.
 |
 |  rsplit(...)
 |      S.rsplit([sep [,maxsplit]]) -> list of strings
 |
 |      Return a list of the words in the string S, using sep as the
 |      delimiter string, starting at the end of the string and working
 |      to the front.  If maxsplit is given, at most maxsplit splits are
 |      done. If sep is not specified or is None, any whitespace string
 |      is a separator.
 |
 |  rstrip(...)
 |      S.rstrip([chars]) -> string or unicode
 |
 |      Return a copy of the string S with trailing whitespace removed.
 |      If chars is given and not None, remove characters in chars instead.
 |      If chars is unicode, S will be converted to unicode before stripping
```

```
 |
 |  split(...)
 |      S.split([sep [,maxsplit]]) -> list of strings
 |
 |      Return a list of the words in the string S, using sep as the
 |      delimiter string.  If maxsplit is given, at most maxsplit
 |      splits are done. If sep is not specified or is None, any
 |      whitespace string is a separator and empty strings are removed
 |      from the result.
 |
 |  splitlines(...)
 |      S.splitlines(keepends=False) -> list of strings
 |
 |      Return a list of the lines in S, breaking at line boundaries.
 |      Line breaks are not included in the resulting list unless keepends
 |      is given and true.
 |
 |  startswith(...)
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      prefix can also be a tuple of strings to try.
 |
 |  strip(...)
 |      S.strip([chars]) -> string or unicode
 |
 |      Return a copy of the string S with leading and trailing
 |      whitespace removed.
 |      If chars is given and not None, remove characters in chars instead.
 |      If chars is unicode, S will be converted to unicode before stripping
 |
 |  swapcase(...)
 |      S.swapcase() -> string
 |
 |      Return a copy of the string S with uppercase characters
 |      converted to lowercase and vice versa.
 |
 |  title(...)
 |      S.title() -> string
 |
 |      Return a titlecased version of S, i.e. words start with uppercase
 |      characters, all remaining cased characters have lowercase.
 |
 |  translate(...)
 |      S.translate(table [,deletechars]) -> string
 |
 |      Return a copy of the string S, where all characters occurring
 |      in the optional argument deletechars are removed, and the
 |      remaining characters have been mapped through the given
 |      translation table, which must be a string of length 256 or None.
 |      If the table argument is None, no translation is applied and
 |      the operation simply removes the characters in deletechars.
```

```
 |
 |  upper(...)
 |      S.upper() -> string
 |
 |      Return a copy of the string S converted to uppercase.
 |
 |  zfill(...)
 |      S.zfill(width) -> string
 |
 |      Pad a numeric string S with zeros on the left, to fill a field
 |      of the specified width.  The string S is never truncated.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

We have therefore plenty of functions ("methods") in the str type ("class"). Let's try a few of them with a classic exercise – find a substring in a given string:

```
In [16]: a=raw_input("Type the first string: ")
         b=raw_input("Type the second string: ")
         print b in a
```

```
Type the first string: test
Type the second string: test
True
```

As you can see, this specific function, _contains_, can also be invoked as an operator **in**. This is a general pattern in object-oriented languages, so that we will be able to employ operators between objects of types that are not the typical numeric ones:

```
In [29]: print 1+2
```

```
3
```

```
In [31]: print "a"+"b", "a"*3
```

```
ab aaa
```

In these examples, we have seen arithmetic operators applied to strings.