言語まとめ Python

[Python][言語 まとめ][Python サンプルコード][Python 標準ライブラリ概観]

参照

■公式

・Python http://www.python.org/doc/

■言語リファレンス

・http://docs.python.org/ref/ref.html
・http://www.python.jp/doc/release/ref/

■チュートリアル

・http://docs.python.org/tut/tut.html
・http://www.python.jp/doc/release/tut/

環境

対話モード

■プロンプト

```
>>>
```

■継続行

```
...
```

■最後に印字された式は変数 _ に代入される

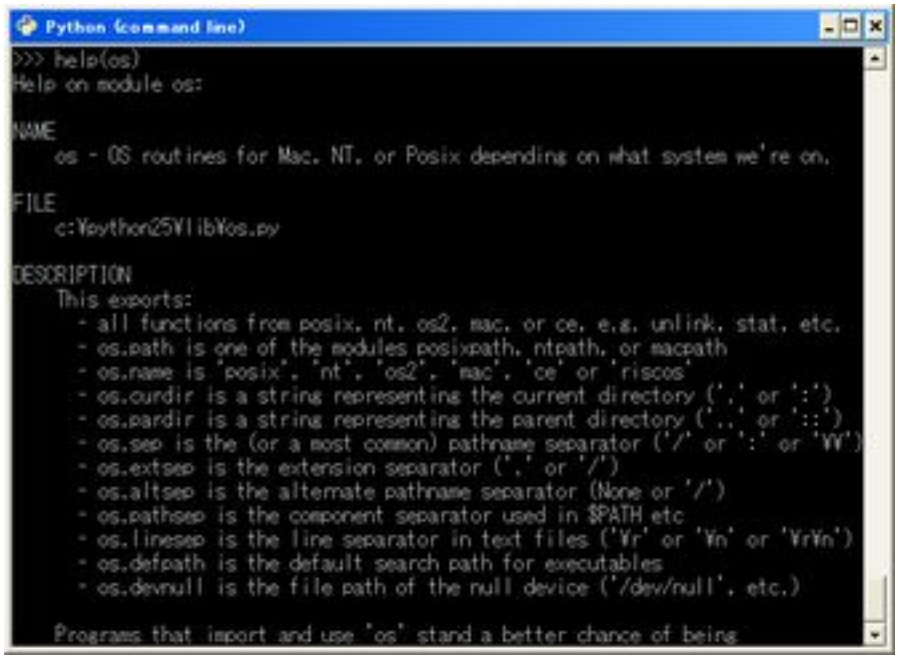
```
>>> 2+2
4
>>> _
4
```

HELP

・help 関数の使い方

■例

```
>>>help(os)
>>>help('string')
```



デバッガ

```
>>> def readfile(f):
      fp = open(f,'r')
        for l in fp:
          print l

>>> import pdb
>>> pdb.run("readfile(r'/home/piroto/test.txt')")
> <string>(1)<module>()->None
(Pdb) step
--Call--
> <pyshell#50>(1)readfile()
(Pdb) args
f = /home/piroto/test.txt
(Pdb)
```

言語基本

- Python はインタプリタ言語です。
- とてもコンパクトで読みやすいプログラム
  - 高レベルのデータ型によって、複雑な操作を一つの実行文で表現
  - 実行文のグループ化はグループの開始や終了の括弧を使う代わりにインデントで行う
  - 変数や引数の宣言が不要
- 拡張する
  - C 言語でプログラムを書く方法を知っているなら、新たな組み込み関数やモジュールをインタプリタに追加することは簡単

対話モード

```
$ python
Python 2.4.4 (#1, Oct 23 2006, 13:58:00)
[GCC 4.1.1 20061011 (Red Hat 4.1.1-30)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello python"
hello python
>>>
```

Python スクリプトの実行

■ python01.py

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
print "hello python"
```

env Wikipedia より

この例では、/usr/bin/env は env コマンドのフルパスである。環境は変わらない。この場合、python インタプリタのフルパスを与えることでもインタプリタを指定できる。この方法の問題は異なるコンピュータシステム上では、パスが異なるかもしれないということである。この例のように代わりに env を使うことで、スクリプトの実行時にインタプリタが検索され場所が特定される。これによりスクリプトがより移植性の高いものとなる。しかし、実行可能ファイルの検索パスにあるすべてのディレクトリの中からマッチするのが検索されるので、違うインタプリタが選択される危険性が高くなる。

coding

ASCII 形式でない文字コードのエンコーディングをソースコードファイル中で使う最良の方法は、#! 行の直後に二行かそれ以上の特殊なコメントを挿入して、ソースファイルのエンコードを指定する

```
# -*- coding: { エンコーディング } -*-
```

■実行

```
# python python01.py
hello python
```

■実行可能にする

```
$ chmod +x python01.py
$ ./python01.py
hello python
```

モジュール

・モジュールとインポート

■サーチパス

1. プログラムのホームディレクトリ
2. 環境変数 PYTHONPATH
3. 標準ライブラリモジュールのディレクトリ
4..pth ファイルの内容

上記 4 つを連結したものが、sys.path となる

```
>>> import sys
>>> sys.path
['', 'C:\\Windows\\system32\\python26.zip', 'C:\\Python26\\DLLs', 'C:\\Python26\\lib', 'C:\\Python26
\\lib\\plat-win', 'C:\\Python26\\lib\\lib-tk', 'C:\\Python26', 'C:\\Python26\\lib\\site-packages']
```

■利用可能なモジュールの一覧

```
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

BaseHTTPServer      anydbm              imageop             sgmllib
Bastion             array               imaplib             sha
CGIHTTPServer       ast                 imghdr              shelve
Canvas              asynchat            imp                 shlex
                      :
```

文の要素

ステートメント

・複合ステートメント (他のステートメントがネストしたステートメント) では、必ず見出し行の末尾にコロン (:) をつける
・行の終わりがステートメントの終わり
・インデントの終わりがブロックの終わり
・同じブロックに属するコードではインデントの仕方を統一する必要がある
・括弧、文末のセミコロンは基本的に不要 (あっても動作はする)

```
>>> x = 1
>>> y = 2
>>> if x > y:
...     print x
... else:
...     print y
...
2
```

■特殊なケース

1 行に複数のステートメントを記述

・セミコロンを利用すると、複合ステートメント以外は 1 行に書ける

```
>>> l = [1, 2]
>>> if l:
...     l[0] = 3; l[1] = 4
...
>>> l
[3, 4]
```

1 ステートメントを複数行にまたがらせる

・リストやディクショナリ、タプルの場合、括弧が閉じるまでは 1 ステートメントとみなされるため、途中で改行してもよい

```
>>> l = [1,
...      2,
...      3]
>>> l
[1, 2, 3]
```

・バックスラッシュで行継続が可能 (現在非推奨)

```
>>> x = 1 + 2 + 3 + \
...     4 + 5 + 6
```

```
>>> x
21
```

■ステートメント一覧

ステートメント	機能
代入	リファレンスの作成
呼び出し	関数の実行
print	オブジェクトの出力
if/elif/else	分岐
for/else	シーケンスの繰り返し処理
while/else	汎用ループ
pass	何もしない
break、continue	ループの飛び越し
try/except/finally	例外の検知
raise	例外の発生
import、from	モジュールへのアクセス
def、return、yield	関数の作成
class	オブジェクトの作成
global	グローバル変数の指定
del	リファレンスの削除
exec	文字列を Python コードとして実行
assert	デバッグ用コードの挿入
with/as	コンテキストマネージャ (2.6 から)

コメント

で始まり、行末まで

エンコード宣言

http://www.python.jp/doc/release/ref/encodings.html

・最初の行か、二行目に以下のようなコメント

```
# -*- coding: <encoding-name> -*-
```

行の継続

■明示的

\ を使う

```
if a and b \
  anc c and d
      return 1
```

・コメントは継続できない
・\ の後にコメントは記述できない

■非明示的

http://www.python.jp/doc/release/ref/implicit-joining.html

・括弧 "()[]{}" 内の式は \ を使わずに分割可能。コメントもつけられる

```
al = ['a', 'b', 'c',  #comment 1
      'd', 'e', 'f',  #comment 2
      'g', 'h', 'i']  #comment 3
```

インデント

http://www.python.jp/doc/release/ref/indentation.html

・インデントレベルは、実行文のグループ化方法を決定するために用いられます。
・先頭の空白 (スペースおよびタブ) の連なりは、その行のインデントレベルを計算するために使われます。
・タブは (左から右の方向に) 1 つから 8 つのスペースで置き換えられ、置き換え後の文字列の終わりの位置までの文字数が 8 の倍数になるように調整されます。

キーワード

http://www.python.jp/doc/release/ref/keywords.html

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

代入ステートメント

```
>>> x = 1                      # 基本形
>>> a, b, c = 'a', 'b', 'c'    # タプル代入
>>> [d, e] = ['d', 'e']        # リスト代入
>>> f, g, h = 'fgh'            # シーケンス代入
>>> i = j = 'ij'               # マルチターゲット
>>> j += 'k'                   # 拡張代入
```

・上記例では、変数 a 〜 h までは、どれも対応する 'a' 〜 'h' が代入される。
・i、j は 同時に 'ij' が代入され、j は 拡張代入により、'ijk' となる

関数定義

def

■関数定義は def で行う

```
>>> def max(i, j):
...     if i > j:
...         return i
...     else:
...         return j
...
>>> print max(10,99)
99
```

docstring

■関数本体の最初の文として文字列リテラルを利用することにより、関数の説明を行う

```
>>> def max(i, j):
...     """ return max value. """
...     if i > j:
...         return i
...     else:
...         return j
...
>>> help(max)
Help on function max in module __main__:

max(i, j)
    return max value.
```

引数

■引数のデフォルト

```
>>> def greet(msg='hello'):
...     print '%s world.' % (msg)
...
>>> greet()
hello world.
>>> greet('good bye')
good bye world.
```

■名前つき引数

```
>>> def profile(name='secret', age='secret'):
...     print 'name: %s, age: %s' % (name, age)
...
>>> profile(age='37')
name: secret, age: 37
>>> profile('Yagi')
name: Yagi, age: secret
```

■名前つき引数をディクショナリとして受け取る

・最後の仮引数が、** 引数名 となっていると、名前つき引数をディクショナリとして受け取る

```
>>> def profile(**prof):
...     print 'name: %s, age: %s' % (prof['name'], prof['age'])
...
>>> profile(name='Yagi',age='37')
name: Yagi, age: 37
```

■ディクショナリを展開して名前つき引数として渡す

・ディクショナリを ** をつけて関数に渡すと、名前つき引数に展開される

```
>>> profile(name='Yagi',age='37')
name: Yagi, age: 37
>>> def profile(name, age):
...     print 'name: %s, age:%s' % (name, age)
...
>>> prof = {'name':'Yagi','age':'37'}
>>> profile(**prof)
name: Yagi, age:37
```

■タプルやリストを展開して引数として渡す

```
>>> parm = (5, 0, -1)
>>> for i in range(*parm):
...     print i
...
5
4
3
2
1
```

■可変引数

```
>>> def var_prms( *prms ):
...     for p in prms:
...         print p
...
>>> var_prms('a', 'b', 1, 2)
a
b
1
2
```

・このような引数はタプルとして引き渡される

```
>>> def var_prms2( *prms ):
...     print prms
...
>>> var_prms2('a', 'b', 1, 2)
('a', 'b', 1, 2)
```

■引数の型チェック

```
>>> def test(word):
  assert isinstance(word, basestring),"word must be string"
  print word

>>> test("abc")
abc
>>> test(123)

Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    test(123)
  File "<pyshell#13>", line 2, in test
    assert isinstance(word, basestring),"word must be string"
AssertionError: word must be string
```

■関数を引数として渡す

```
>>> def test(func,data):
  print func(data)

>>> test(len,range(10))
10
>>> test(sum,range(10))
45
```

lambda 式を渡す

```
>>> test(lambda n : n * n, 5)
25
```

無名関数 lambda (ラムダ) 式

[Scheme]

・定義と使用が 1 文で済む、無名関数を書くことができる

```
>>> inc = lambda x : x + 1
>>> x = 0
>>> while x < 5:
...     print x
...     x = inc(x)
...
0
1
2
3
4
```

■ラムダ式で条件式を使う

・text の大文字小文字を反転させ、順序を逆にして返す

```
rev = lambda x : x.upper() if x.islower() else x.lower()
l = [rev(x) for x in text]
l.reverse()
return ''.join(l)
```

ジェネレータ関数

■ yield

・yield 文は、ジェネレータ関数を定義するときだけ使われる。また、yield を使うだけで、関数はジェネレータ関数になる。
・ジェネレータ関数が呼び出されると、ジェネレータを返す。ジェネレータの next() が例外を発行するまで繰り返し呼び出して実行する。
・yield 文が実行されると、ジェネレータの状態は凍結 (全てのローカルな状態が保存) され、次に next() が呼び出された際に利用できる。

例

```
def gen_seq(s, e):
    x = s
    while True:
        yield x
        x += 1
        if x > e:
            raise StopIteration

for i in gen_seq(2, 5):
    print i
```

結果

```
2
3
4
5
```

高階関数

・高階関数とは、関数を引数にしたり、関数を戻り値とするような関数のこと。

■ filter 関数を使う

・filter は要素から関数の結果が True となる値を抽出

奇数を返すフィルタ

```
>>> def odd(x):
...     return (x % 2) != 0
...
>>> l = range(10)
>>> filter(odd,l)
[1, 3, 5, 7, 9]
```

■ map 関数を使う

・map 要素を関数に渡した戻値からなるリストを返す

```
>>> def suqare(x):
...     return x * x
...
>>> l = range(10)
>>> map(suqare, l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

■リストの内包表記

・map() や filter() を使用せずに、リストを生成する
・式、for 節、後ろに続くゼロ個かそれ以上の for 節または if 節からなる

小文字のリストを大文字のリストに変換

```
>>> items = ['aaa', 'bbb', 'ccc']
>>> [item.upper() for item in items]
['AAA', 'BBB', 'CCC']
```

数値のリストを自乗のリストに変換

```
>>> n = [1, 2, 3, 4, 5]
>>> [ x * x for x in n ]
[1, 4, 9, 16, 25]
```

条件に合う辞書を作成する

```
>>> m = {'a':1,'b':2,'c':1,'d':2}
>>> m2 = dict([(k, v) for k, v in m.iteritems() if v == 1])
>>> m2
{'a': 1, 'c': 1}
```

ビルトイン型

ビルトイン型の分類

オブジェクトの型	カテゴリ	上書き

数値	数値	不可
文字列	シーケンス	不可
リスト	シーケンス	可
ディクショナリ	写像	可
タプル	シーケンス	不可
ファイル	エクステンション	該当せず

・Python では、数値オブジェクトが上書き不可のため、インクリメント (++)、デクリメント (--) 演算子が使用できない。

n 進数

■ 16 進数

```
>>> 0xff
255
>>> hex(255)
'0xff'
```

■ 8 進数

```
>>> 010
8
>>> oct(8)
'010'
```

■文字列数値変換

```
>>> int('11111111',2)
255
>>> int('0xff',16)
255
```

演算子

比較演算子

すべてのオブジェクトによりサポートされる

演算子	内容	備考
<	小なり	
<=	小なりイコール	
>	大なり	
>=	大なりイコール	
==	等号 (オブジェクトの内容が同等であるか)	
!=	等号否定	推奨
<>	等号否定	時代遅れ

is	オブジェクトが同一であるか	
is not	オブジェクト同一性否定	

オブジェクト比較の例

```
>>> l1 = [1, 2, 3]
>>> l2 = l1          # リファレンスをコピーするため、l2 は l1 と同一オブジェクトを参照
>>> l3 = l1.copy()   # 内容をコピーするため、l3 は l1 と内容は同一の別オブジェクトとなる
>>> l3 = l1[:]
>>> l1 == l2         # 内容は同一
True
>>> l1 == l3         # 内容は同一
True
>>> l1 is l2         # 実体も同一
True
>>> l1 is l3         # 実体は別
False
```

大きさを比較する式を複数組み合わせることができる

```
>>> x, y, z = 1 ,2 ,3
>>> x < y < z
True
```

論理演算子

演算子	備考
x or y	ショートサーキット
x and y	ショートサーキット
not x	not は Boolean 以外の演算子より優先度が低い not a b は not (a b) と同意

数値

数値変換

■整数

・文字列や数値を整数に変換する組み込み関数

```
int()
```

■浮動小数点数

・文字列や数値を浮動小数点数に変換する組み込み関数

```
float()
```

真偽値

真偽値	整数値

True	1
False	0

文字列

・文字列処理メソッドのまとめ

文字列はシングルまたはダブルのクォートで囲む

```
msg1 = "message1"
msg2 = 'message2'
```

エスケープ (\n: 改行文字) と継続 (\ で文の継続)

```
>>> m = "m1\n\
... m2\n\
... m3"
>>> print m
m1
m2
m3
```

raw 文字列

・文字列リテラルを ``raw" 文字列にすると、\n のようなエスケープシーケンスは改行に変換されません

```
>>> m = r"hello\nworld"
>>> print m
hello\nworld
```

・ユニコード文字列を扱う場合、ur とする。

ヒアドキュメント

・対になった三重クォート """ または ''' で文字列を囲むこともできます。三重クォートを使っているときには、行末をエスケープする必要はありません、しかし、行末の改行文字も文字列に含まれることになります。

```
>>> print """line1
... line2
... line3"""
line1
line2
line3
```

文字列の連結 (+)、反復 (*)

・文字列は + 演算子で連結させる (くっつけて一つにする) ことができ、 * 演算子で反復させることができます。

```
>>> msg1 = "a"
>>> msg2 = "b"
>>> print msg1 + (msg2 * 3)
abbb
```

セパレータで文字列を結合 : join() メソッド

```
>>> l = ['f1', 'f2', 'f3']
>>> m = ','.join(l)
>>> m
'f1,f2,f3'
```

文字の切り出し

■インデクス表記

・文字列は添字表記 (インデクス表記) することができます

```
>>> msg1 = "abcde"
>>> print msg1[2]
c
```

■スライス表記

・部分文字列を スライス表記 : コロンで区切られた二つのインデクスで指定することができます。

```
>>> num1 = "12345"
>>> print num1[1:3]
23
```

・インデクスを負の数にして、右から数える

```
>>> msg = "012345"
>>> msg[-1] # 右端
'5'
>>> msg[-2] # 右端から 2 桁目
'4'
>>> msg[-2:] # 右端から 2 桁目以降
'45'
>>> msg[:-2] # 右端から 2 桁目まで
'0123'
```

文字列長 : len() 関数

・組込み関数 len() は文字列の長さを返す

```
>>> len(msg)
6
```

ユニコード

・Unicode オブジェクトを利用

```
>>> msg = u"矢木"
>>> msg
u'\u77e2\u6728'
>>> print msg
矢木
```

・Python ファイルの文字コード

書式操作

・http://www.python.jp/doc/release/lib/typesseq-strings.html
・Python 書式

変数

命名ルール

・先頭は英文字あるいは下線、その後は英文字、数字、下線
・大文字と小文字は区別
・予約語は変数名に使用できない

命名の慣例

・先頭に下線を 1 つ付けた名前は通常使わない。(from module import * ステートメントでインポートできないため)
・先頭と末尾に下線を 2 つ重ねた名前は使わない。(システム定義の特別な名前として利用されることが多いため)
・先頭だけに下線を 2 つ重ねた名前は通常使わない。(使用した場合、所属クラスの名前をつけた変数名に自動変換される。マングリングという)
・下線 1 つだけの変数名は使用しない。(対話型コマンドラインで、直前の実行結果を保持するのに使用される)
・大文字で始まる名前は通常クラス名として使用
・モジュール名は小文字
・self は予約語ではないが、特別な意味を持つので変数名として使用しない
・ビルトインクラスの名前は予約語ではないが、変数名として使用しない

グローバル変数

用例

```
def init_env():
    global use_decimal
    global initial_guess

    use_decimal = True
    initial_guess = 10.0
```

演算

順次

pass

・pass はなにもしない。構造的に文が必要な場合に、何もする必要がない場合

```
>>> v = None
>>> if v == None:
...     pass
... else:
...     print 'do something, when v != None.'
...
```

選択

if...elif...else

```
x = int(raw_input("please input int:"))
if x < 0:
    print "negative"
elif x == 0:
    print "zero"
else:
    print "positive"
```

・空ではない文字列やリストは、真と評価される

```
>>> l = ['','abc']
>>> if l:
      print 'list is not empty'
list is not empty
>>> for i in l:
      if i:
        print 'item[%s] is not empty' % (i)
item[abc] is not empty
```

繰り返し

for

Python では、for は、コレクションのイテレーターとして使う

```
>>> alp = ['a','b','c','d']
>>> for a in alp:
...     print a
...
a
b
c
d
```

■指定回数繰り返す : range() 関数

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in range(10):
...     print i
...
0
1
2
3
4
5
6
7
8
9
```

■指定回数インデックスを降順に繰り返す : range() 関数

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

```
>>> for i in range(9, 0, -1):
...     print i
...
9
8
```

```
7
6
5
4
3
2
1
```

■インデックスと要素を同時に取得 : enumerate() 関数

リストで使う

```
>>> itms = ['a','b','c']
>>> for i, v in enumerate(itms):
...     print '%d:%s' % (i, v)
...
0:a
1:b
2:c
```

辞書の場合

・iteritems() メソッドを使うと、キーとそれに対応する値を同時に取り出せる

```
>>> m = {'a':1,'b':2,'c':1,'d':2}
>>> for k, v in m.iteritems():
...     print '%s,%s' % (k, v)
...
a,1
c,1
b,2
d,2
```

■要素を組で取得 : zip() 関数

```
>>> l1 = ['a','b','c']
>>> l2 = ['1','2','3','4']
>>> for v1, v2 in zip(l1, l2):
...     print '%s:%s' % (v1, v2)
...
a:1
b:2
c:3
```

while

```
>>> while i < 10:
...     print i
...     i += 1
...
0
1
2
3
4
5
6
7
8
9
```

break

```
>>> for i in range(10):
```

```
...     if i == 3:
...         break;
...     print i
...
0
1
2
```

continue

```
>>> for i in range(10):
...     if i % 2 == 0:
...         continue
...     print i
...
1
3
5
7
9
```

else(繰り返しでの)

■リスト終了時に呼び出される

```
>>> for i in range(3):
...     print i
... else:
...     print 'finished loop by range'
...
0
1
2
finished loop by range
```

■ break での終了では PDFJ::Text=HASH(0x9457618)

```
>>> for i in range(3):
...     if i == 1:
...             break
...     print i
... else:
...     print 'break!'
...
0
```

データ構造

リスト型

■用法

・コンマで区切られた値からなるリストを各カッコで囲む
・要素をすべて同じ型にする必要はない

インデックス、スライス、連結なども可能

```
>>> lst = ['a', 'b', 3, 4]
>>> lst
['a', 'b', 3, 4]
>>> lst[0]
'a'
>>> lst[-2]
```

```
3
>>> lst[1:3]
['b', 3]
```

代入、スライスに対する代入も可能

```
>>> lst[1:3] = ['c']
>>> lst
['a', 'c', 4]
```

リストのリスト

```
>>> lst[1] = [1,2,3]
>>> lst
['a', [1, 2, 3], 4]
```

■メソッド

メソッド	内容
append(x)	末尾に要素を追加
extend(L)	指定したリストの要素を対象のリストに追加
insert(i, x)	指定位置に要素を挿入します。インデックスを持つ要素の直前に挿入
remove(x)	値 x を持つ最初の要素を削除。存在しなければ ValueError
pop([i])	指定された位置にある要素をリストから削除し、その要素を返す。インデックスが指定なしなら末尾の要素が対象
index(x)	値 x を持つ最初の要素のインデックスを返す。存在しなければ ValueError
count(x)	x の出現回数
sort()	内容自体をソート。
reverse()	内容自体を逆順ソート。

```
>>> l = ['a', 'b', 'c']
>>> l.append('d')
>>> l
['a', 'b', 'c', 'd']
>>> l2 = ['e','f','g']
>>> l.extend(l2)
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> l.insert(2,'b')
>>> l
['a', 'b', 'b', 'c', 'd', 'e', 'f', 'g']
>>> l.remove('b')
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> l.pop()
'g'
>>> l
['a', 'b', 'c', 'd', 'e', 'f']
>>> l.index('b')
1
```

```
>>> l.count('c')
1
>>> l.reverse()
>>> l
['f', 'e', 'd', 'c', 'b', 'a']
>>> l.sort()
>>> l
['a', 'b', 'c', 'd', 'e', 'f']
```

■リストをスタックとして利用する

```
>>> stack = []
>>> for i in range(5):
...     stack.append(i)
...
>>> for i in range(5):
...     print stack.pop()
...
4
3
2
1
0
```

■リストをキューとして利用する

```
>>> que = []
>>> for i in range(5):
...     que.append(i)
...
>>> for i in range(5):
...     print que.pop(0)
...
0
1
2
3
4
```

■コンパレータを利用したソート

```
def comparator(x, y):
       if x == y: return 0
       if x  < y: return -1
       return 1

sort_area.sort(cmp=comparator,reverse=True)
```

■リストを連結して文字列を作成

```
>>> l = ['hello','python','world']
>>> ' '.join(l)
'hello python world'
```

■リストに含まれる値のインデックスを得る

```
>>> l = ['a','b','c','c']
>>> l.index('c')
2
```

シーケンスに対する反復処理

表記法	内容
for item in s	s のアイテムを反復処理

for item in sorted(s)	s のアイテムをソートして反復処理
for item in set(s)	s の異なるアイテムを反復処理 (set 化)
for item in reversed(s)	s の要素を逆順に反復処理
for item in set(s).difference(t)	s の中で t に存在しない要素を反復処理
for item in random.shuffle(s)	s の要素をランダムな順序で反復処理

リストの削除

■スライスで部分削除

・Python スライス表記

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del l[1:9]
>>> l
[0, 9]
```

■全要素を削除

```
>>> del l[:]
>>> l
[]
```

■リスト自体を削除

```
>>> del l
>>> l
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'l' is not defined
```

例外

ステートメント

ステートメント	内容
try / except	例外の検知と処理
try / except / else	例外が発生しない場合、else ブロックを処理
try / finally	例外の有無にかかわらずクリーンアップ処理
raise	例外を発生させる
assert	一定の条件が満たされたときに例外を発生させる
with / as	コンテキストマネージャに関連

例

■ try / except

```
>>> l = [0,1,2]
>>> try:
...     print l[3]
... except IndexError as e:
...     print 'IndexError'
...
IndexError
```

■ raise

```
>>> class Bad:
...     pass
...
>>> try:
...     raise Bad()
... except Bad:
...     print 'got Bad'
...
got Bad
```

■ try / except / else / finally

・else は例外が発生しなかった場合のみ実行される
・finally は例外の有無にかかわらず実行される

```
>>> l = [0,1,2]
>>> try:
...     print l[1]
... except IndexError:
...     print 'got exception'
... else:
...     print 'not exception'
... finally:
...     print 'finally process'
...
1
not exception
finally process
```

TODO

オブジェクト

・データを抽象的に表したもの
・データは全て、オブジェクトまたはオブジェクト間の関係
・アイデンティティ値 (identity) 、型 (type) 、そして値 (value) を持つ

アイデンティティ値

・一度生成されると、アイデンティティ値 は決して変化することがありません
・演算子 is は、二つのオブジェクト間のアイデンティティ値を比較
・関数 id() は、オブジェクトのアイデンティティ値を表す整数 (現在の実装ではオブジェクトのメモリ上のアドレス) を返す

型

・型 もまた一度生成されると、変わることがない
・type() 関数は、オブジェクトの型 (型自体も一つのオブジェクトです) を返す

値

・オブジェクトによっては、 値 (value) を変えることができる
・値を変えることができるオブジェクトは 変更可能 (mutable) であるという
・値を一度設定すると、その後は変えることができないオブジェクトは 変更不能 (immutable) であるという

ガベージコレクション

・オブジェクトを明示的に破壊することはできません ; しかし、オブジェクトに到達不能 (unreachable) になると、ガベージコレクション (garbage-collection) によって処理されます。実装では、ごみ収集を遅らせたり、全く行わないようにすることができます

標準型

Python に組み込まれている型

型	内容
None	値が存在しないことをしめす
NotImplemented	被演算子が該当する演算を行うための実装をもたない場合、この値を返すことがある
Ellipsis	スライス内に "..." 構文がある場合に使われます
モジュール (module)	モジュールは import 文で import します
クラス	クラスオブジェクトはクラス定義で生成されます
クラスインスタンス	クラスインスタンスはクラスオブジェクトを呼び出して生成します

・Numbers
  ・整数型 (integer)

型	内容
(通常の) 整数型 (plain integer)	-2147483648 から 2147483647 までの整数を表現。演算の結果が定義域を超えた値になった場合、結果は通常長整数で返される。
長整数型 (long integer)	無限の定義域を持ち、利用可能な (仮想) メモリサイズの制限のみをうける。
ブール型 (boolean)	False または True を表現。整数のサブタイプで、ほとんどの演算コンテキストにおいてブール型値はそれぞれ 0 または 1 のように振舞う。
型	内容
浮動小数点数型 (floating point number)	浮動小数点数を表現 Python は単精度の浮動小数点数をサポートしません
複素数型 (complex number)	浮動小数点を 2 つ一組にして複素数を表現

・シーケンス型 (sequence)
・変更不能なシーケンス (immutable sequence)

型	内容
文字列型 (string)	各要素は文字 (character) 文字型 (character type) は存在しません
Unicode 文字列型	各要素は Unicode コード単位
タプル型 (tuple)	タプルの要素は任意の Python オブジェクトにできます

・変更可能なシーケンス型 (mutable sequence)

型	内容
リスト型 (list)	要素は任意の Python オブジェクトにできます

・マップ型 (mapping)

型	内容
辞書型 (dictionary)	ほとんどどんな値でもインデクスとして使えるような、有限個のオブジェクトからなる集合を表す

・呼び出し可能型 (callable type)

型	内容
ユーザ定義関数 (user-defined function)	関数定義を行うことで生成
ユーザ定義メソッド (user-defined method)	クラスやクラスインスタンス (あるいは None) を任意の呼び出し可能オブジェクト (通常はユーザ定義関数) と結合し (combine) ます
ジェネレータ関数 (generator function)	呼び出された際に、常にイテレータオブジェクトを返します。このイテレータオブジェクトは関数の本体を実行するために用いられます
組み込み関数 (built-in function)	組み込み関数オブジェクトは C 関数へのラッパ
組み込みメソッド (built-in method)	実際には組み込み関数を別の形で隠蔽したもの
クラス型 (class type)	クラス型オブジェクトは通常、そのクラスの新たなインスタンスを生成する際のファクトリクラスとして振舞いますが、 new() をオーバライドして、バリエーションを持たせることもできます。呼び出しの際に使われた引数は new() に渡され、さらに典型的な場合では新たなインスタンスを初期化するために init() に渡されます。

旧クラス型 (classic class)	クラスオブジェクトが呼び出されると、新たにクラスインスタンスが生成され、返されます。この操作には、クラスの init() メソッドの呼び出し (定義されている場合) が含まれています。
クラスインスタンス (class instance)	ラスインスタンスはクラスが call() メソッドを持っている場合にのみ呼び出すことができます。 x(arguments) とすると、 x.call(arguments) 呼び出しを短く書けます。

・内部型 (internal type)

型	内容
コードオブジェクト	バイトコンパイルされた (byte-compiled) 実行可能な Python コード、別名 バイトコード (bytecode) を表現します。
フレーム (frame) オブジェクト	実行フレーム (execution frame) を表します。実行フレームはトレースバックオブジェクト内に出現します
トレースバック (traceback) オブジェクト	例外のスタックトレースを表現します
スライス (slice) オブジェクト	拡張スライス構文 (extended slice syntax) が使われた際にスライスを表現するために使われます。拡張スライス構文とは、二つのコロンや、コンマで区切られた複数のスライスや省略符号 (ellipse) を使ったスライスで、例えば a[i:j:step] 、a[i:j, k:l] 、あるいは a[..., i:j] です。スライスオブジェクトは組み込み関数 slice() で生成されます
静的メソッド (static method) オブジェクト	関数オブジェクトからメソッドオブジェクトへの変換を阻止するための方法を提供します。通常はユーザ定義メソッドオブジェクトを包むラッパです。
クラスメソッドオブジェクト	別のオブジェクトを包むラッパであり、そのオブジェクトをクラスやクラスインスタンスから取り出す方法を代替します。