Python入門 - 関数

関数(def)

def 文を用いて**関数(function)**を定義することができます。下記の例では、x と y という二つの引数を受け取り、その合計値を表示する関数 add() を定義し、それを呼び出しています。

Python
```
def add(x, y):
    print x + y

add(3, 5)           #=> 8
```

return は関数の戻り値を指定します。return 文を省略すると、関数は **None** を返します。

Python
```
def add(x, y):
    ans = x + y
    return ans

n = add(3, 5)
print n             #=> 8
```

下記の様に、キーワード付き引数を指定することができます。キーワード付き引数は関数定義側で、デフォルトの値を指定することができます。

Python
```
def repeat_msg(msg, repeat=3):
    for i in range(repeat):
        print msg

repeat_msg('Hello')                 # Hello, Hello, Hello
repeat_msg('Yahho', repeat=5)       # Yahho, Yahho, Yahho, Yahho, Yahho
```

*name は残りの順序引数を、**name はキーワード付き引数を辞書型で受け取ります。

Python
```
def func(a1, a2, *args, **params):
    print a1                #=> A
    print a2                #=> B
    print args               #=> ('C', 'D')
    print params              #=> {'k1': 'K1', 'k2': 'K2'}


func('A', 'B', 'C', 'D', k1='K1', k2='K2')
```

args や params を変数で渡す場合には、呼び出す側にも * や ** をつけます。

Python
```
args = ('C', 'D')
params = {'k1': 'K1', 'k2': 'K2'}
func('A', 'B', *args, **params)    #=> A, B, ('C', 'D'), {'k1': 'K1', 'k2': 'K2'}
```

関数は、複数の値を返却することができます。

Python
```
def func():
    return 3, "ABC"

n, s = func()
print n, s              #=> 3 ABC
```

関数定義の冒頭には、"""..."""で[ドキュメントストリング]を記述することができます。

Python
```
def func(x, y):
    """A sample function"""
    return x + y
```

グローバル変数(global)

関数の外部で定義された変数は **グローバル変数** として扱われます。関数の中でグローバル変数を参照することはできますが、代入することはできません。代入する場合は、**global** で宣言する必要があります。

Python
```
count = 0               # グローバル変数

def func():
    print count         # 参照することはできる
    global count        # global宣言してやれば
    count += 1          # 代入することもできる
```

globals() はグローバル変数、**locals()** はローカル変数の一覧を辞書として返却します。

Python
```
def func():
    for k in globals().keys():
        print "GLOBAL: %s = %s" % (k, globals()[k])
    for k in locals().keys():
        print "LOCAL: %s = %s" % (k, locals()[k])

func()
```

ラムダ式(lambda)

ラムダ式(lambda)式は、名前のない小さな関数を定義します。ラムダ式自体は式として扱われるため、関数の引数に指定することができます。

Python
```
myfunc = lambda x, y: x + y
print myfunc(3, 5)              #=> 8
```

lambda式は、sorted(), map(), filter()などの関数に渡す無名関数として利用されることがあります。

Python
```
a = [1, 2, 3]
print map(lambda x: x ** 2, a)      #=> [1, 4, 9]
```

イテレータ(iterator)

イテレータ は for文 で使用することができる繰り返し機能を持つオブジェクトです。イテレータオブジェクトは、**__iter__()** で next() メソッドを持つオブジェクトを返却し、**next()** メソッドは次の要素を返却し、最後に達すると **StopIteration**例外を返すようにします。

Python
```
class MyClass:
    def __init__(self):
        self.data = (1, 2, 3, 4, 5)
        self.index = 0
    def __iter__(self):
        return self
    def next(self):
        if self.index < len(self.data):
            self.index += 1
            return self.data[self.index - 1]
        else:
            raise StopIteration

for n in MyClass():
    print n                 #=> 1, 2, 3, 4, 5
```

最後の for文は、下記と同等の動作を行っています。

Python

```
it = MyClass().__iter__()
while 1:
    n = it.next()
    try:
        print n                 #=> 1, 2, 3, 4, 5
    except StopIteration:
        break
```

ジェネレータ(yield)

yield は、**イテレータ**を返却する**ジェネレータ**を定義する際に用いられます。まずは、yield を使用しない例を見てみます。

Python
```
def funcA(list):
    ret = []
    for n in list:
        ret.append(n * 2)
    return ret

for n in funcA([1, 2, 3, 4, 5]):
    print n                     #=> 2, 4, 6, 8, 10
```

上記を yield によるイテレータを用いた方式に変更すると下記になります。どちらも実行結果は変わりませんが、funcA() が [2, 4, 6, 8, 10] のリストを返却するのに対し、funcB() はイテレータオブジェクトを返却します。イテレータの場合、値が参照される度に次の値が計算されて返却されます。この例では、5個程度のリストなのでパフォーマンスの差異はありませんが、100万個のリストの場合、funcA() は関数を呼び出した時点で100万個のリストを生成してしまうのに対し、イテレータの funcB() の場合は、利用した際にのみ次の値を計算するので、メモリや処理効率が改善されます。

Python
```
def funcB(list):
    for n in list:
        yield n * 2

for n in funcB([1, 2, 3, 4, 5]):
    print n                     #=> 2, 4, 6, 8, 10
```

もう少し例を見てみましょう。下記は、ファイルを読み込み、__END__ が出現するまでの行を表示するプログラムです。

Python
```
def readfileA(f):
    lines = []
    for line in f:
        lines.append(line.rstrip())
    return lines

f = open("test.txt")
for line in readfileA(f):
    if (line == "__END__"):
        break
    print line
f.close()
```

yield によるイテレータを用いると下記の様になります。上記が、最初にファイルをすべて読み込んでメモリに展開してしまうのに対して、イテレータを使用した例では、1行分のメモリしか使用せず、__END__ がきた時点で残りは読み飛ばして処理を終了することができます。

Python
```
def readfileB(f):
    for line in f:
        yield line.rstrip()

f = open("test.txt")
for line in readfileB(f):
    if (line == "__END__"):
        break
    print line
f.close()
```

デコレータ(@)

関数を実行する前後に特殊な処理を実行したい場合、**@デコレータ** を用いることができます。下記の例では、hello()関数を

mydecolater でデコレート(装飾)しています。デコレーション関数では、関数実行前に start を、関数実行後に end を出力しています。

Python

```
def mydecolater(func):      # デコレータを定義する
    def wrapper():
        print "start"       # 前処理を実行する
        func()              # デコレート対象の関数を実行する
        print "end"         # 後処理を実行する
    return wrapper

@mydecolater
def hello():
    print "hello"

hello()                     #=> start, hello, end
```

デコレータをもう少し汎用化したのが下記のサンプルです。ターゲット関数に @decolater を付与することで、関数の引数や戻り値を表示するサンプルです。wraps()は、ターゲット関数の関数名やドキュメントストリング(__doc__)を保持するために呼び出します。

Python

```
def mydecolater(func):
    import functools
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print "Funcname:", func.__name__
        print "Arguments:", args
        print "Keywords:", kwargs
        ret = func(*args, **kwargs)
        print "Return:", ret
        return ret
    return wrapper

@mydecolater
def func(msg1, msg2, flag=1, mode=2):
    """A sample function"""
    print "----", msg1, msg2, "----"
    return 1234

n = func("Hello", "Hello2", flag=1)
print n

print repr(func)
print func.__doc__
```

初版：2014年12月30日、最終更新：2017年11月26日
http://www.tohoho-web.com/python/function.html