オブジェクト指向プログラミング

第９回

箕原辰夫

関数の定義と利用

- 関数の定義の仕方
- 定義された関数の利用の仕方
- 引数（パラメータ）ありの関数
- オプションの引数・キーワード引数
- グローバル変数とローカル変数
- 値を戻す関数の定義
- 再帰関数の定義
- 高階関数・内部関数

関数を定義する理由

- 例：多重の繰返しで、プログラムの一部分が一体何をやっているのかわからなくなってきた
  - 意味のあるブロックに名前をつけて、外に出して、それを呼び出すようにする。

    drawSineCurveなど
- 例：少しだけ異なる（例えば一辺の長さあるいは角度が違うだけ）がほぼ同じ処理をしている部分がある。
  - その機能に名前を付けて、異なるデータをパラメータで受け渡して、プログラムを構造化する

関数の２つの局面

- 関数を定義する
  - **def**構文を用いて定義する。

    def drawPaint(c) : ....

- 関数を呼び出す
  - これは、いままでも散々やってきました。

    print("Hello, Python Programming")

関数の定義（記述）

```
def 関数名( ):
    その関数が呼ばれたら行なわせたい内容
```

- 行なわせたい内容は、for文などと同様にブロックの形にしてインデントを下げる
- あるいは、１行で記述できる場合は、コロン以降の同じ行に書くことも可能である。
- 関数名は、小文字始まりで動詞を使うのが一般的
- 動詞＋名詞→　setRectangle　のように名詞を大文字にする（Javaの流儀：Camel case）
  set_rectangle（Pythonで良く使われる命名法：Snake case）
  setrectangle（ドイツ語のような命名法：lower case）

関数の定義の例

```
def displayNumber( ):
    for n in range( 1, 11 ):
        print( n, end =  " " )
    print( )


  def shortSleep( ): print( "ZZZ…" )
```

関数の呼出し

関数名( )　　あるいは

オブジェクト名.関数名( )

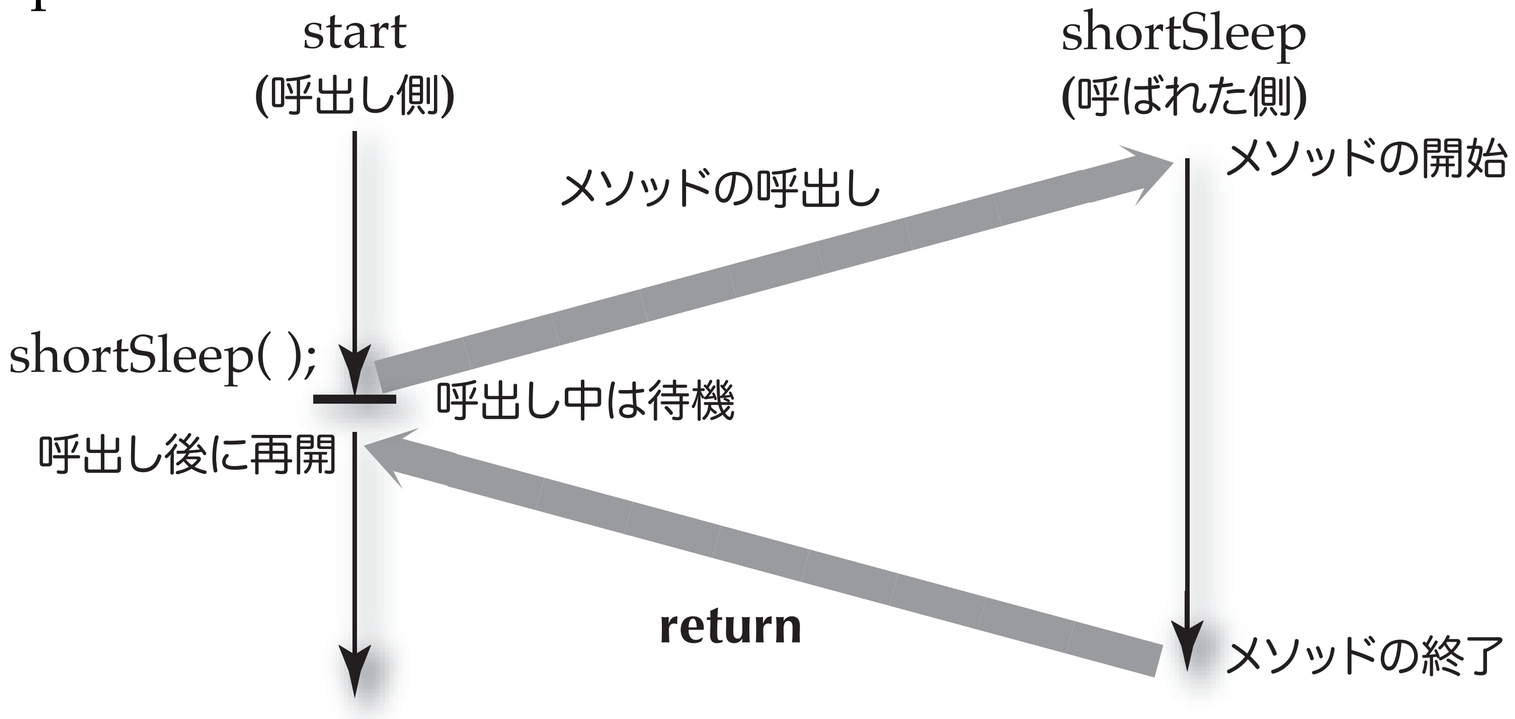
定義された関数であれば、関数名だけで呼び出せる

displayNumber( )

shortSleep( )

関数の呼出しの構造

- shortSleepの場合

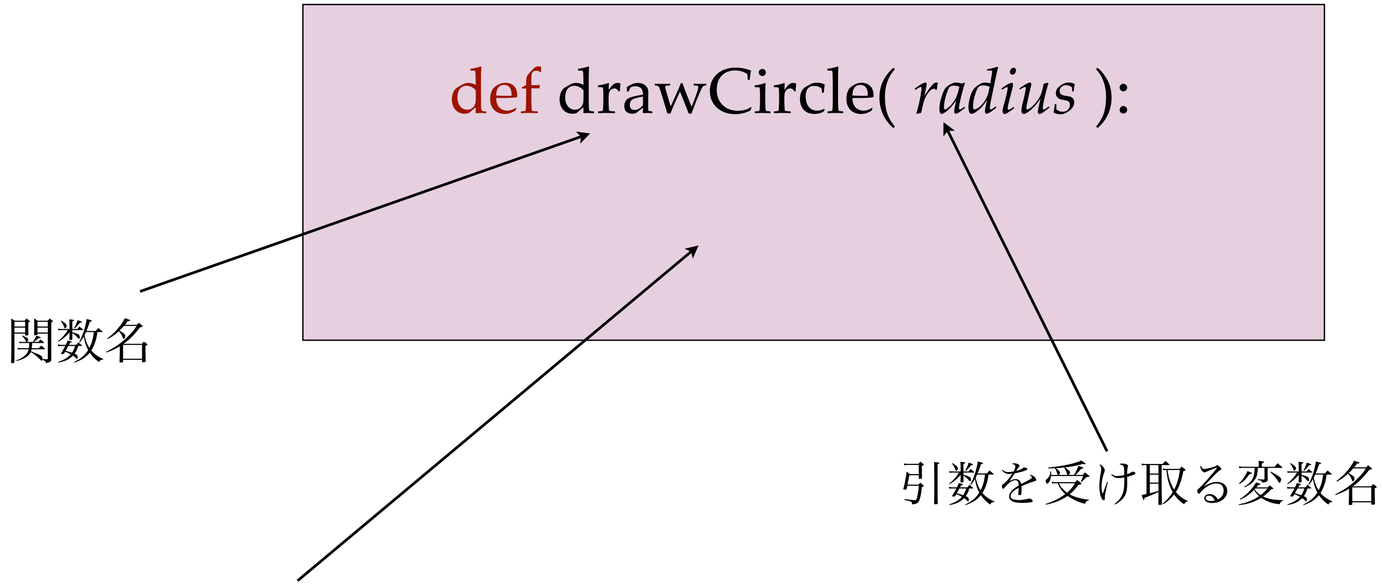


Pythonの特有の関数定義の法則

- 関数は、呼び出される前には定義しておく必要がある。
- 同じ名前の関数を再定義することができる。呼び出すときには、最後に定義された関数が呼び出される。

引数のある関数の定義

def 関数名(　変数名 [, 変数名]…) :
　関数で行なわせたい内容



引数のある関数の呼出し

関数名(実引数の式 [, 実引数の式]…)

drawCircle(34 * x)

まずこの部分が計算される

先ほどの*radius*に代入される

仮引数と実引数

- 引数…argument, パラメータ…parameter
- 仮引数（仮パラメータ）
  - 関数の定義で宣言されている変数のこと
- 実引数（実パラメータ）
  - 関数を呼び出すときに、呼び出し側で与えられる式のこと。この式がまず評価されて、定数値（あるいはオブジェクトを指す値）になってから、仮引数に代入されて、該当の関数が呼び出される。

引数のある関数の例

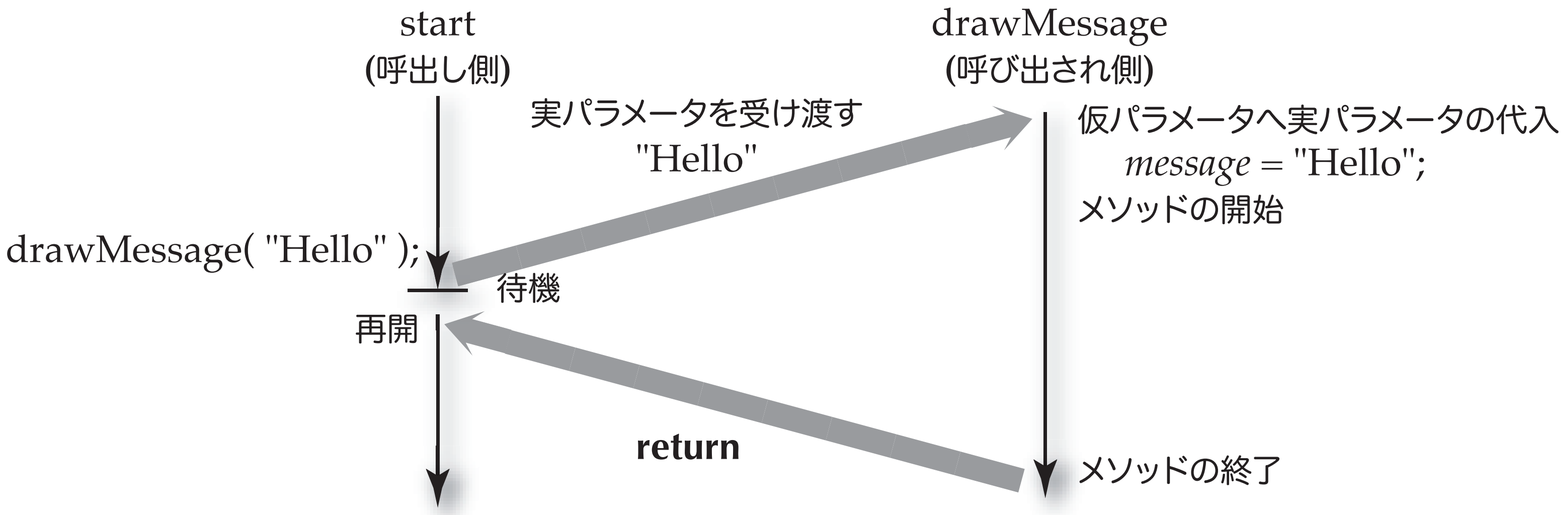
- 改行しないで表示を行なうdrawMessageの例

```
# drawMessageの定義
def drawMessage( message ):
    print( "Message is ", message, end="" )

# drawMessageを呼び出す
drawMessage( "Hello" )
```

引数のある関数の呼出しの構造

- drawMessageの場合



複数の引数

- 複数の仮引数があるときは、実引数が順番に仮引数に代入される（位置引数：positional argument）

```
def  multiply( a, b ):
     print( a * b )

multiply( 12, 3 )  #  12がaに、3がbに代入される
```

オプションの引数（Python特有）

- 関数の定義で、引数のところに=をつけて、その引数が省略された場合のデフォルト値を指定しておける
- 例：

```
def displayPerson( name="John", gender="male" ):
  print( "Name:", name, " Gender:", gender )
```

- オプションの引数の場合は、キーワードの引数名を指定することで、順番に関係なく、その引数に代入することができる。また、実引数を省略しても構わない。
- 例：

```
displayPerson( )
displayPerson( gender="female", name="Mary" )
displayPerson( name="Ken" )
displayPerson( "Susanna", "female" )
displayPerson( "Smith" )
```

引数の個数を可変にした関数の定義

- 書式：
  - **def** 関数名（ *変数名 ）:

    関数の処理の定義

- 関数の処理の定義の中で
  - 変数名はタプルとして使える

- また、「*変数名」の記法を使って、タプルを展開してアクセスすることが可能である。

- 例：

```
def sample( *message ):
    # *変数名で受け取る
    for n in message:    print( n )
    # 個別の要素をアクセスしたい場合
    print( message )
    # そのままタプルとして渡される
    print( *message )
    # 個々の要素が展開される
```

- 呼出し例：

```
sample( 12, 324, "ABC", 22.5, 5+4j )
```

可変個数の引数とオプション引数を使う場合

- 順序で代入される引数（位置引数）は、最初に定義しておく
- 可変個数の引数を定義しておく
- 一番最後に、オプション引数を定義する

- 例：

```
def displayPerson( name, *child,  gender="male" ):
 print( name, gender, *child, sep=":" )
```

キーワードのある可変個数の引数を取る関数

- オプションのキーワードと値の対を可変個数持つ関数を定義することができる
- 書式：

  def 関数名(**可変引数名):

    関数の本体

- 例：

```
def printKeywords( **keywords ):
    print( keywords )  #  辞書構造になっているのがわかる
    print( *keywords )  # キーの一覧だけを取り出せる
    otherfunc( ** keywords ) # すべて展開される

# 展開されたオプション引数を持つ関数
def otherfunc( name="", age=0 ):
    print( name, age )

printKeywords( name="John", age=23 )  # 呼び出してみる
```

位置引数の指定（Python 3.8より）

- 仮引数を指定するときに、/記号を使って、その前は位置引数として指定する記法ができた
- *は、そこからキーワード（オプション）引数が始まることを示す
- キーワード引数は、オプション引数でもあるが、必ず関数呼出しのときに、キーワードを指定しなければならない
- 書式：
  - **def** 関数名(位置引数, ..., /, オプション引数, ..., *, キーワード引数, ...):
- 例：

  def g(a, b, /, c, d, *, e, f): print(a, b, c, d, e, f) # 定義例

  g(10, 20, 30, d=40, e=50, f=60) # この呼出しははOK

  g(10, b=20, c=30, d=40, e=50, f=60) # bはデフォルト値を持つ引数ではだめ

  g(10, 20, 30, 40, 50, f=60) # eをキーワード引数にしないとだめ

- なお、/の前にあるのは位置引数になるため、キーワード引数に同じ名前の変数を利用することも可能となった
- 例：

  def f(a, b, /, **kwargs): print(a, b, kwargs) # 定義例

  f(10, 20, a=1, b=2, c=3) # aとbは、キーワード引数の変数名としても利用されている

  出力結果は、以下のようになる

  10 20 {'a': 1, 'b': 2, 'c': 3}

関数の構造化と呼出し

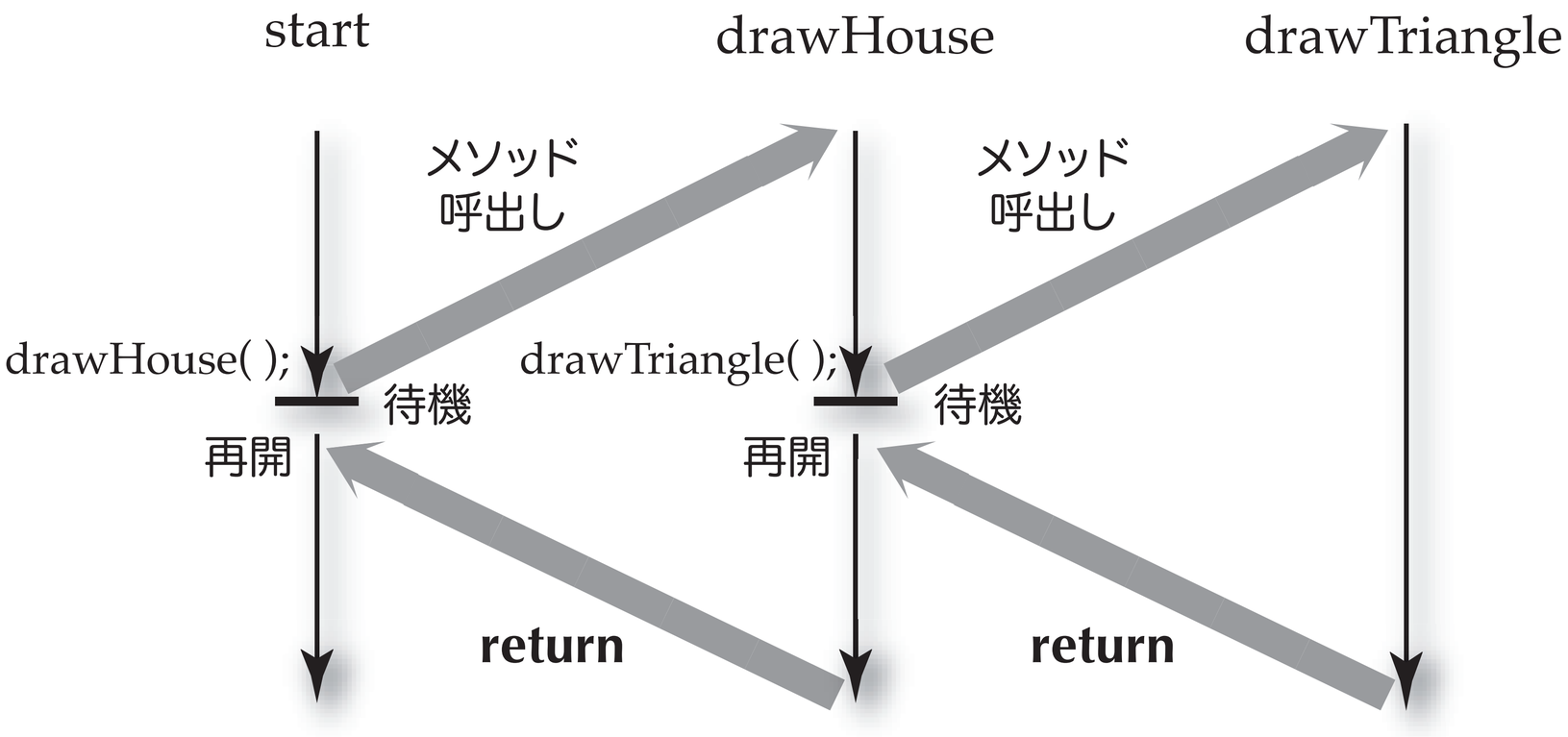
- 一連の細かな作業を１つの機能として定義したい。
- それぞれの細かな作業はそれぞれ既に関数として定義されている。その間を調整したい。
  - それらの関数を呼び出す新たな機能を持つ関数を定義する
  - 最初は、新たに定義されたその関数を呼び出す。
  - ボトムアップ・アプローチ（Bottom-up Approach）

関数の構造化の意義

- 大きな作業を 1 つの関数として定義する。
- その作業の一部を実現してくれるような関数を更に新たな関数として定義していく
- 大きな問題から、小さな問題に分割していく方法は、段階的詳細化（stepwise refinement）あるいはトップダウン設計（top-down design approach）と呼ばれる
- 逆に小さな機能の集合の要素を利用して、より複雑な機能に集約していく方式をボトムアップ設計（bottom-up design approach）と呼ぶ。
- 上記のどちらかの手法を用いるプログラミング手法を構造化プログラミング（structured programming）と呼ぶ。

多重の関数呼出しの構造

- drawHouseの場合



グローバル変数とローカル変数

- グローバル変数（大域変数）はすべての関数で参照可能
- 関数の中だけで使われるローカル変数（局所変数）は、関数の実行と共に消える

```
x=20 # Global Variable

def sample( ):
   y=30 # Local Variable
```

ローカル変数によるグローバル変数の隠蔽

- グローバル変数と同じ名前のローカル変数を使うことができる。
- その関数の中では、ローカル変数が優先され、グローバル変数は、隠蔽されてしまう。
- 同じ名前のローカル変数に代入してもグローバル変数の値は書き変わらない

```
x=20 # Global Variable

def sample( ):
   x=30 # Local Variable with the same name
```

関数の中でのグローバル変数の書換え

- 関数の中でグローバル変数を書き換えたければ、global文で、その変数はグローバルであるという指定を行なう

  書式 **global** 変数名, 変数名, …

```
x = 20
def sample( ) :
    global  x
    x = 30

sample( ) # 実行後は、xの値は30になる
```

隠蔽されたグローバル変数にアクセスする方法

- globals()…使えるグローバル変数の一覧が辞書（dict）構造で返ってくる
  - globals( )["変数名"]…グローバル変数に直接アクセスできる
  - 変数の値を書き換えることも可能になっている

- locals( )…その関数内で使えるローカル変数の一覧が辞書構造で返ってくる
  - locals( )["変数名"]…変数にアクセスできる
  - 変数の値を書き換えることはできない

引数で受渡し vs グローバル変数

- 最初に 1 回だけ設定して、後は参照だけするような情報、あるいは対象が1つしかない情報（オブジェクト）は、グローバル変数でも良い。

```
# 共通で使うグローバル変数

c = Canvas( win, width=500, height=500 )
# 関数から、cでアクセス

def paintCircle( ):
    c.create_circle( 100, 100, 50 )
```

グローバル変数とローカル変数

- globals()とlocals()組込み関数は、関数の実行中に呼び出すと、そのときの利用できる変数と値の一覧を見ることができる（辞書構造になっている）

```
def sample( ):   # 関数定義
  z = 12   # ローカル変数を関数内で使用
  print( "ローカル変数: ", locals() )
  print( "グローバル変数:", globals() )
```

値を戻す関数

- **return**文を使う。呼出し側に値を戻せる。関数の処理も終了する。
  - 関数の処理だけを終わらせる場合は、**return**と記述する
  - 値を戻す場合は、関数の中で、
    - **return** 式
    - 例： **return** 34 * 32

- 他のプログラミング言語と異なり、複数の値を戻すこともできる
  - 例： **def** multiReply( ) :
    return 12, 13, 14

squareとgreater

- # 与えられた値の２乗を返す関数
- **def** square (x) :
- 　　**return** x ** 2

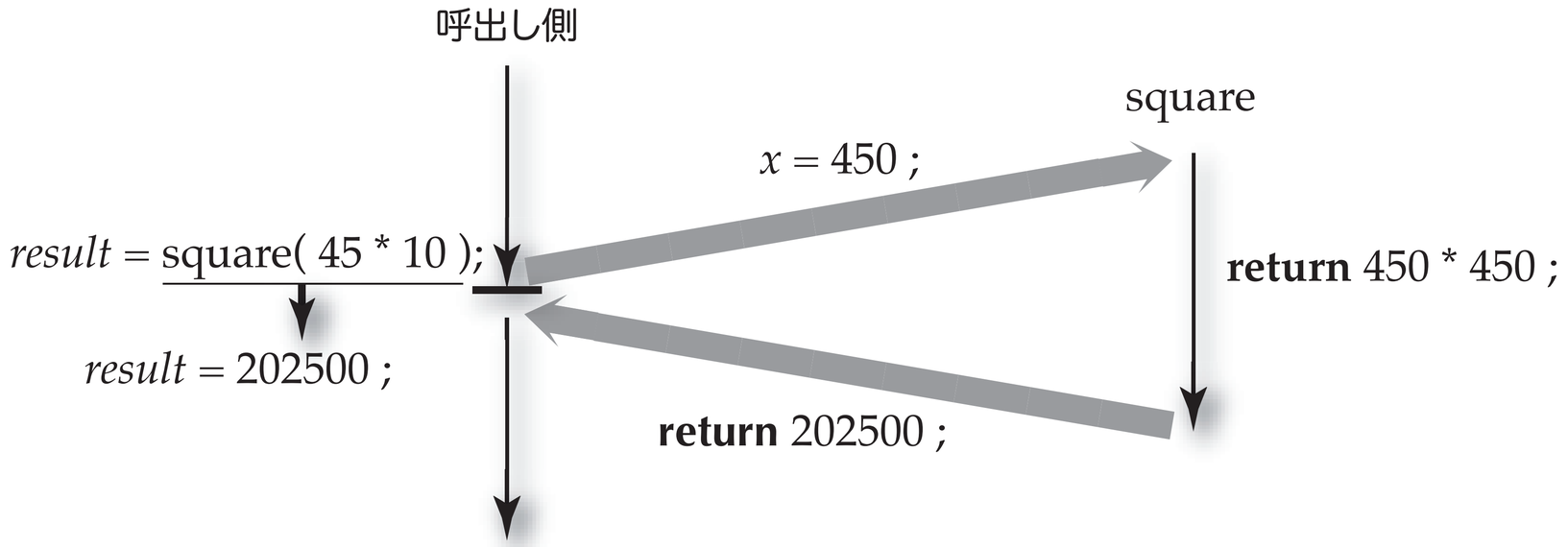
- # ２つのうち、大きいほうの値を返す関数
- # max(x, y)と同じ
- **def** greater(x, y):
- 　　**if** x > y : **return** x
- 　　**else**: **return** y

- **def** greater(x, y): **return** x **if** x > y **else** y

値を戻す関数の呼出し

- 変数に代入する式の中で呼び出される。
- まず実パラメータが評価され、呼び出された後に関数の仮引数に代入される。

➡例： *result* = square(45 * 10)



値を戻す関数を多重に呼出し

- 他のパラメータ付き関数の呼出しの際に、実パラメータの中で呼び出される

- 例: *result* = square(greater(34, 56))
  - まず、34, 56の２つパラメータでgreaterが呼び出される。
  - 返ってきた値を実パラメータとして（この場合は56）、squareが呼び出される
  - 返ってきた値が*result*に代入される（3136）

複数の値を返す関数の呼出し

- 受け取る側でも、複数の変数を用意し、カンマで区切って代入する

- **def** getMaxMin(a, b):
  return max(a, b), min(a, b)

- more, lesser = getMaxMin(56, 89) # more, lesser = 89, 56
- values = getMaxMin(34, 56) # values = (56, 34)

引数や関数の戻り値の型指定（Python 3.6より）

- 関数の仮引数や戻り値（return value）の型アノテーション、型ヒントが標準として導入された
- 実際には、特に型をチェックする訳ではない
- なお、リストは[要素の型名], タプルは(要素の型名,), 辞書は[キーの型名, 値の型名]で記述する
- 書式
  - **def** 関数名(仮引数名: 型) -> 戻り値の型:
  - 例：

```
def sample( i_list: [ int ], n : int ): pass
def sample( i_tup:  (int,) , n : int ) -> float: return  89.6
def sample( i_dict: [ int, str ], n : int ) -> str: return
"ABC"
```

- リストやタプルの要素の型や戻り値の型がどれでも良いときには、typingパッケージに入っているAnyで記述する

```
from typing import Any
def sample( a:[], b:(Any,) ) -> Any: return 10
```

- 型チェックする訳ではないので、実際に呼び出す際にあっていなくても構わない
- 例：

```
def conv( a : int ) -> str : return 12   # 整数をもらって、文字列を返すつもり
print( conv( "aaa" ) )  # 特に文句は言われない
```

型チェックをするツール

- mypy：Pythonの静的（実行時前に行なう）型チェッカーで、型アノテーションをチェックします。--strictオプションを使用すると、より厳密なチェックが可能です。
  - pip install mypy
  - mypy --strict your_script.py
- Pyright：Microsoftが開発した静的型チェッカーで、VS Codeの拡張機能としても利用できます。
  - pip install pyright
  - pyright your_script.py
- strictly：ランタイムで型アノテーションを厳密にチェックするためのツールです。strictly.require_hintsフラグを使用すると、すべての引数に型ヒントが必要になります。
  - pip install strictly
  - import strictly

    @strictly.enforce

    def my_function(a: int, b: str) -> bool:

      return isinstance(a, int) **and** isinstance(b, str)
- Microsoft copilotに聞きました。

オブジェクトを返す関数

- 呼び出した側で受け取るための変数が必要

- **def** getRedWindow( ):
  win = Tk( ) # ウィンドウを作る
  win.title("red window") # ウィンドウのタイトル
  win.geometry("500x500+20+20") # 大きさと座標
  win.attributes("-topmost", True) # 最前面表示
  win.configure(background="red") # 背景色
  return win

- red = getRedWindow( )

構造物のリテラルを関数に渡す場合

- 辞書・集合・リスト・タプルなどの構造物を関数に引数として渡す場合は、コピーされたものが渡される訳ではなく、参照（ポインタ）渡しになる
- 例：

```
def updateFirstItem( a ):  a[ 0 ] = a[ 0 ] + 12
nlist = [ 23, 34, 45 ]
updateFirstItem( nlist )
print( nlist[ 0 ] )  # 35が表示される
```

約数と素数

- 約数を求めて表示する関数displayDivisorsを作る
  - 繰返しで、その数まで割り切れる数があったら、表示するようなもの
- 約数の個数を求める関数countDivisorsを作る
- 素数であるかどうかを判定する関数isPrimeを作る
  - 素数とは、１とその数でしか割りきれない数のこと
  - 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, …
  - ある自然数nが、素数かどうか判定するときは、2から$\sqrt{n}$までの数について割り切れるかどうか考えれば良い（Wikipedia「素数」の項参照）

完全数とピタゴラス数、そして合同数

- 完全数…その数を除く約数の和が、その数と等しい
- 例： 6 = 1 + 2 + 3　6の約数は1, 2, 3, 6

- ピタゴラス数…$a^2 + b^2 = c^2$ を満たす自然数の組
  (a, b, c)のこと
- 例：$3^2 + 4^2 = 5^2$　なので　(3, 4, 5)

- 合同数…ピタゴラス数のa, bを使って、n = a * b / 2の数

- 完全数とピタゴラス数、合同数を表示する関数を作成してみる

合同数（congruent）

- 直角三角形の面積となるような数
- 以下を満たすようなnが合同数である（https://ja.wikipedia.org/wiki/合同数）

$$a^2 + b^2 = c^2$$

$$\frac{ab}{2} = n$$

- 整数の合同数を求めてみる

- 信用できないやり方：
- 一部の合同数は、以下のように求めることもできる。p を奇数の素数（奇素数）とする。
  - p を 8 で割ったあまりが 3 のとき、p は合同数ではなく、2p は合同数である。
  - p を 8 で割ったあまりが 5 のとき、p は合同数である。
  - p を 8 で割ったあまりが 7 のとき、p と 2p は合同数である。

2進数と完全数

- Wikipediaの「完全数」の項目参照
- 完全数6, 28などを2進数で表わしてみると、110, 11100というように、1の前後に1と0が同数つくような形になっている
- この1...1の部分は、2^p-1ということでメルセンヌ数と呼ばれている。10進数になおすと、$11_{(2)}=3=2^2-1$ とか $111_{(2)}=7=2^3-1$ など。
- また、10....の部分は、2^{p-1} で表わされる。
- つまり、完全数の候補は、$\left(2^p-1\right)\left(2^{p-1}\right)$ で表わされることになる。
- メルセンヌ数が素数であった場合（メルセンヌ素数と呼ばれる）、この完全数の候補は、完全数であることが知られている

素因数分解

- 与えられた数nを素数の積として分解する
- 2〜n/2までの間で、素数かどうかを判定する
- 素数だったら、nをその素数で割り切れる間は、割り続ける
- nが1になったら終了

再帰呼出し（recursion）

- 関数の中から、その関数自身を呼び出す
- 引数を利用する関数で実現することができる
- 次のような方法でプログラミングする
  - １．基底レベルでの処理を記述

    ✳そのレベルでは、もう再帰呼出しをしない
  - ２．それ以上のレベルでの処理を記述

    ✳再帰呼出し（レベルを下げて呼び出す）および、その前後の処理を記述
- 基底レベルがないとプログラムが止まらなくなる

値を戻す関数と再帰

- 階乗を計算するfactorial
- n! = 1 × 2 × 3 × … × n-1 × n
  - n == 1のときは、1を返す
  - n > 1のときは、n * factorial(n-1)を返す
  - **def** factorial(n): **return** 1 **if** n <= 1 **else** n * factorial(n-1)
- 総和を計算するsummation
- $\Sigma_{i=1}^{n}$ i = 0 + 1 + 2 + 3 + … + n-1 + n
  - n == 0のときは、0を返す
  - n > 0 のときは、n + summation(n-1)を返す
  - **def** summation(n): **return** 0 **if** n <= 0 **else** n + summation(n-1)
- 一番最後に再帰呼出しをするのをtail recursionと呼ぶ

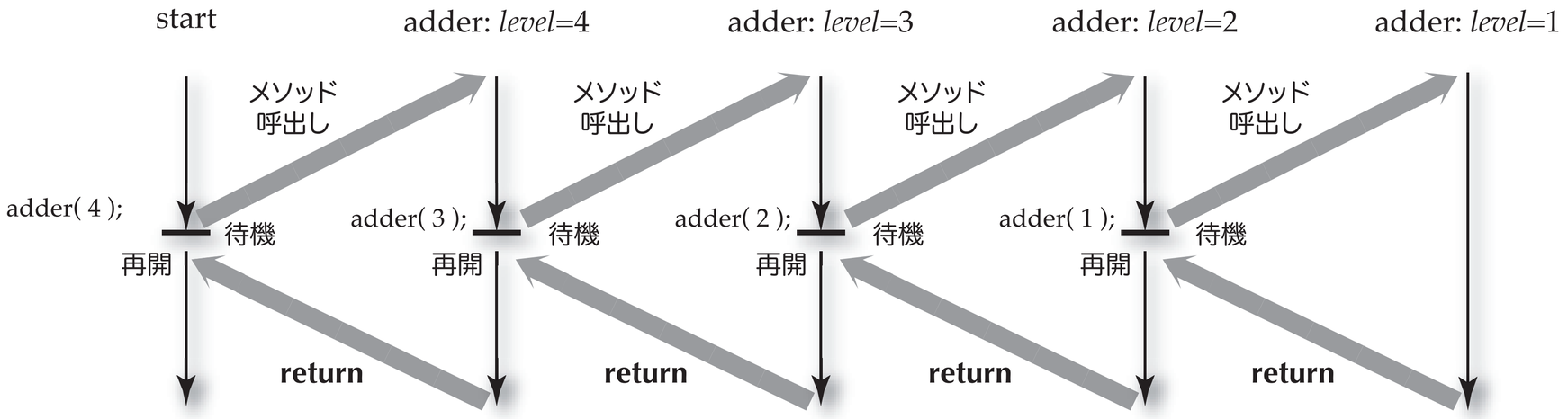
元について

- 元：その数と演算したときに、もともとの数が求まる数

- 掛け算の元＝1
  - n * 1 = n

- 足し算の元＝0
  - n + 0 = n

再帰呼出しの過程


start
adder: level=4
adder: level=3
adder: level=2
adder: level=1
メソッド
呼出し
メソッド
呼出し
メソッド
呼出し
メソッド
呼出し
adder(4);
adder(3);
adder(2);
adder(1);
待機
待機
待機
待機
再開
再開
再開
再開
return
return
return
return

ユークリッドの互除法

- 最大公約数を求める方法
  - ２つの数のうち、大きい方と小さい方に分ける
  - 大きい方の数を小さい方の数で割り切れたら、小さい方の数が求める答え
  - 割り切れなかったら、次の大きい方の数に小さい方の数を代入し、小さい方の数には、「大きい方の数%小さい方の数」を代入して、上記の判定を繰り返す
  - 大きい方の数 % 小さい方の数 （余りを使う方法）
  - 大きい方の数 – 小さい方の数（差を使う方法）
- 余りを使った方が、はやく収束する。

GCDの関数

```
# ２つの数の最大公約数を求める関数（再帰版）

def gcd( n, m ) :
  more, less = max( n, m ), min( n, m )
  if  more % less == 0 : return less
  else : return  gcd( less, more % less )


value = gcd( 356, 248 )
```

べき乗を求める関数

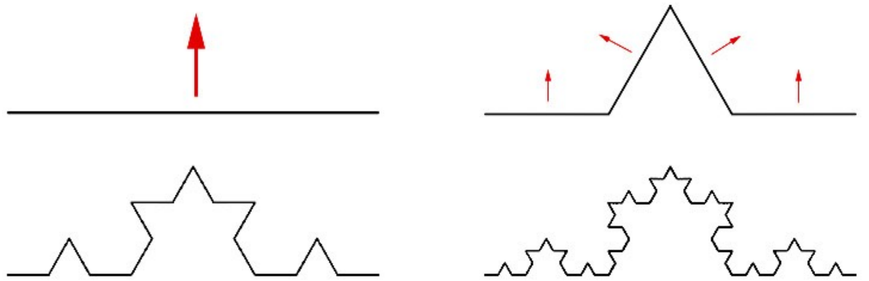
- 他のプログラミング言語では、べき乗を求める演算子がない。それと合わせるために、べき乗をもとめる関数を作ってみる。
- パラメータは、基になる数と、指数

- **def** power(x, n) : # xのn乗を返す

```
result = 1
for i in range( n ):  result *= x
return  result
```

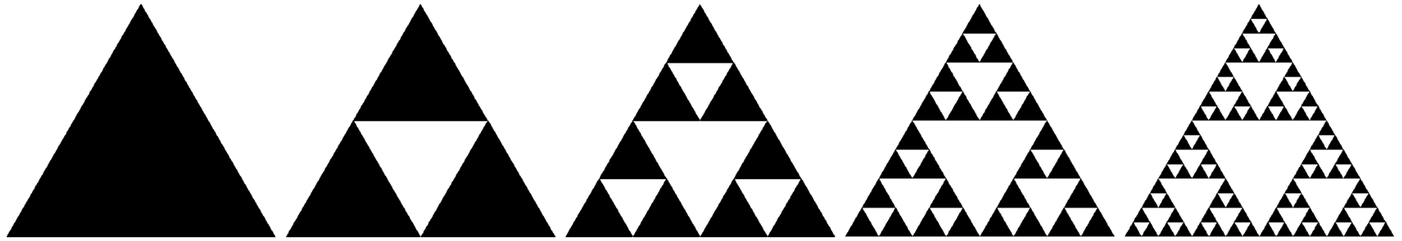
再帰曲線（折れ線）を描く

- コッホ曲線の漸進的な描画方法



ja.wikipedia.orgより

- シェルピンスキーのギャスケット

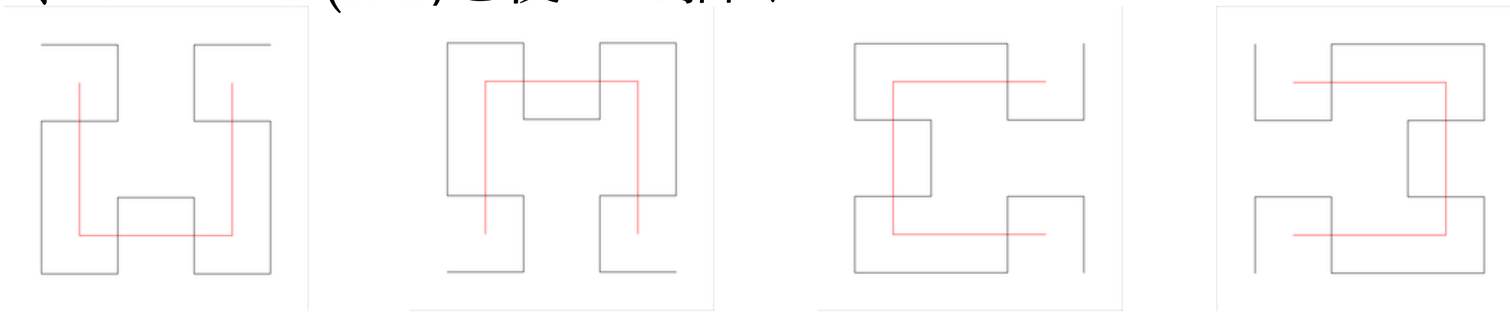


ja.wikipedia.orgより

ヒルベルト曲線

- Wikipedia参照
- レベル１は、４方向ある。東西南北
- タートルグラフィックスを使う場合、右回りと左回りだけ区別すれば良い

- レベル２（n）は、レベル１(n-1)を使って描く



折れ線の方向指定

- **Enum**クラスを使って、列挙型定数を導入（内部的には順序性がある）
- **from** enum **import** Enum
- **class** 列挙名(Enum) :

  名前 = 値

  …
- 特に数値を指定しないようなものに使う
  - **class** Direction(Enum)
    up, down, left, right = "up", "down", "left", "right"
  - **class** DoorStatus(Enum):
    open, closed, halfOpen = "OP", "CL", "HO"
- 変数に代入して、参照、比較などが可能
  - dir = Direction.up
  - if dir == Direction.down: ....

ヒルベルト曲線の描画（タートル版）

- 時計回りと反時計回りの2種類で良い
- レベル１は、それぞれこんな感じ
  - 時計回り　⊐⇒→ɔ↓ɔ←
  - 反時計回り　⊏⇒←ɔ↑ɔ→
- レベル２以降は、最初と最後のタートルの向きをレベル1に合わせてそれを調整する感じ
  以下の記法でɔは右回り90度、ɔは左回り90度を示す
  - 時計回り　ɔ⊏ɔ→⊐ɔ→ɔ⊐→ɔ⊏ɔ
  - 反時計回り　課題にて

回文を作る

- 最初は、任意の文字を選ぶ
- 前後に同じ文字（任意の文字）を追加していく
- これを再帰で行なう

フィボナッチの強欲算法

- フィボナッチの強欲算法のアルゴリズムを用いて、分数を単位分数の和に直して表示する。

$$\frac{x}{y} = \frac{1}{\lceil y/x \rceil} + \frac{x - (y \bmod x)}{y\lceil y/x \rceil}$$

- $\lceil x \rceil$…天井関数（ceil）その数と等しいか、その数よりも大きい最小の整数にする関数（切り上げ）例：$\lceil 4.0 \rceil = 4$, $\lceil 2.7 \rceil = 3$, $\lceil 1.3 \rceil = 2$
- mod…あまり
- このアルゴリズムの部分は、再帰を用いて記述する。
- 強欲算法については、Wikipediaの「エジプト式分数」の項を参照。

高階関数

- 関数のパラメータとして関数を渡すもの。
- 関数の名前を実パラメータ（実引数）として、別の関数に渡す。
- Javaでは関数渡しができないので、オブジェクト渡しで代用。
- 例：

```
def  abc( n ): return "ABC" + str( n )

def drawMessage( fun ): print( fun( 3 ) )

drawMessage( abc )  # "ABC3"
```

tkinter ボタンを使う

- buttonを配置しておく
- buttonのbind高階関数を使って、ボタンが押されたときに呼び出される関数を定義しておく。
- 使用例：

  button=Button(text="OK（了解）", width=50)

  button.bind("<Button-1>", 関数名)

  button.pack()
- "<Button-1>"は、左ボタン

  "<Button-2>"は、ホイールボタン（ホイールクリック）

  "<Button-3>"は、右ボタン
- 呼び出される関数（コールバック関数と呼ぶ）は、eventを仮引数に持つようにしておく
  - 例：

    def processButton(event): print("Button Clicked")
    button = Button(text="Process")
    button.bind("<Button-1>", processButton)

tkinterでマウス入力、キー入力

- マウスは、コールバック関数にevent引数を取り、その引数のx, yの属性に、マウス入力された座標が入る
- キー入力は、コールバック関数のevent引数のchar属性に、入力されたキーの文字が入る。
  keysym属性に、キーの文字列が入る。keycode&0xffで、256未満のキーコードが入る。
- 例：

```
def mouseClicked( event ): print( event.x, event.y )
def keyTyped( event ): print( event.char )

win = Tk()
win.bind( "<Button-1>", mouseClicked )
    # window全体を拾う、キャンバスだけなら、winのところをcにする
win.bind( "<Key>", keyTyped )
```

キー入力

- event変数の中でキー入力の値を調べられる
  - event.keysym…入力されたキーのシンボル（tkinter上で定義されている）の文字列（例："Up", "Down", "Right", "Left", "Control_L", "Shift_R"など）
  - event.keycode…入力されたキーのコード（整数）
  - event.char…入力されたキーの文字
- tkinterのキー入力用のキーのシンボルとコードの一覧
  - https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/key-names.html

tkinterでのcall back関数でのイベント消費

- tkinterでは、マウス入力と、ボタン入力など、同じマウスの入力を貰う関数が複数存在する場合、すべてのコールバック関数が呼ばれる。
  - 例：ボタン入力をマウスで行なった場合、その座標で、マウス入力のコールバック関数も呼ばれる
- 他のコールバック関数にイベントを伝播させずに、その関数で処理を終了させたい場合は、コールバック関数の最後で、"break"の文字列を返すようにする。
  - 例：return "break"

無名関数

- 高階関数への実引数として渡すための名前のついていない（anonymous）関数定義
- 書式：
  - **lambda** 仮引数 : 値を返す式
- これと等価なのは、
  - **def** _(仮引数) : **return** 値を返す式
- 使用例：

```
def drawMessage( fun ): print( fun( 3 ) )
drawMessage( lambda n: "ABC" * n )  #  "ABCABCABC"
```

リストを使った高階関数

- filter(関数, リスト)…関数を各要素に適用してTrueの要素だけ抜き出したリストのオブジェクトを作る
- map(関数, リスト)…各要素に関数を適用した新しいリストのオブジェクトを作る
- functools.reduce(関数, リスト)…各要素の前後に関数を適用して、計算をする
  import functoolsが必要

- sorted(リスト, key=functools.cmp_to_key(比較関数))
- max(リスト, key=functools.cmp_to_key(比較関数))
- min(リスト, key=functools.cmp_to_key(比較関数))

内部関数

- 関数のブロック内部で関数を定義するもの
- global文の替わりにnonlocal文で外側の変数を書き換えることができる
- 例：

```
def sample( ):
  x = 10
  def dummy( ): x = 20; print( x )
  def concrete( ): nonlocal x; x = 30
  dummy( ) ; print( x )
  concrete( ) ; print( x )
```

yield文とコルーチン的に振る舞う関数

- 関数の中にyield文を入れると、値を返すことができるが、関数自身の実行は継続される。
- for文などで繰返し値を受け取ることが可能になる、rangeのように、一度リストに直してしまうと膨大な処理が掛かるところを遅延評価（lazy evaluation）によって、必要になったら値を作り出すことができるようになった
- 例：

```
def numGenerator( ):
    n = 1
    while n < 100:
        yield  n
        n = n * 2 + 1

for m in numGenerator( ):  print( m )
```

- 上の例では、有限のシーケンスになっているが、Python3以降は、無限のシーケンスを持つyield文を作成できるようになった

関数の代入

- lambda式と代入文を使って、関数を定義することができる
- 例：

```
square = lambda n: n ** 2
power = lambda b, p: b ** p
```

- 使用例：
  - square(12) # 144
  - power(3, 4) # 81

- 関数の代入式を用いて、同じ内容の関数を別の名前で定義することも可能である
- 例：

```
sqsq = square
```

- 元の関数名を別に定義した場合、共有した関数名の方は元の定義の内容が残る
- 例：

```
square = lambda n: n ** 0.5
print( sqsq( 2 ), square( 2 ) ) # 4 1.414...
```

partialによる関数の部分適用（カリー化）

- functools.partialを使うと、実引数の一部を与えて、残りの仮引数を持つ関数を作る（関数プログラミング言語においては「カリー化：currying」と呼ばれている）ことができる
- 例：

  ```
  def power( base, expo ) : return base ** expo
  ```

  - 2のべき乗（累乗）を返す、1つの仮引数を持つ関数にする

  ```
  from functools import partial
  binary_power = partial( power, 2 )
  print( binary_power( 9 ) )  # 512
  ```

デコレータ（decorator）

- 関数の定義の前に「@関数名」をつけることができる
- デコレータで指定された関数は、高階関数になっており、定義された関数が実引数になって、ラップすることができる
- 例：

```
@wrapper # デコレータ

def square( x ): return x ** 2
#これは、以下と等価になる

square = wrapper( square ) # squareをラップして再定義
```

- 複数のデコレータがあった場合、一番直前のデコレータから順番に適用されるかたちでラップすることが可能になる

```
@outer
@middle
@inner
def square( x ): return x ** 2

#これは以下と等価になる

square = outer( middle( inner( square ) ) )
```