# スクリプト言語プログラミング
# Pythonによる数値解析

第5回講義資料

箕原辰夫

# 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

  ➡drawRegularPolygon( n, size )など

# 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（lowerCamelCase）

  → set_rectangle　のように間にアンダーバーをいれる（lower_snake_case）

  → setrectangle　のように何も入れない（ドイツ語流）

  → SetRectangle　動詞も名詞も大文字始まり（UpperSnakeCase: C#流）
- 参考：https://mik2062.jp/naming/

# 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( )

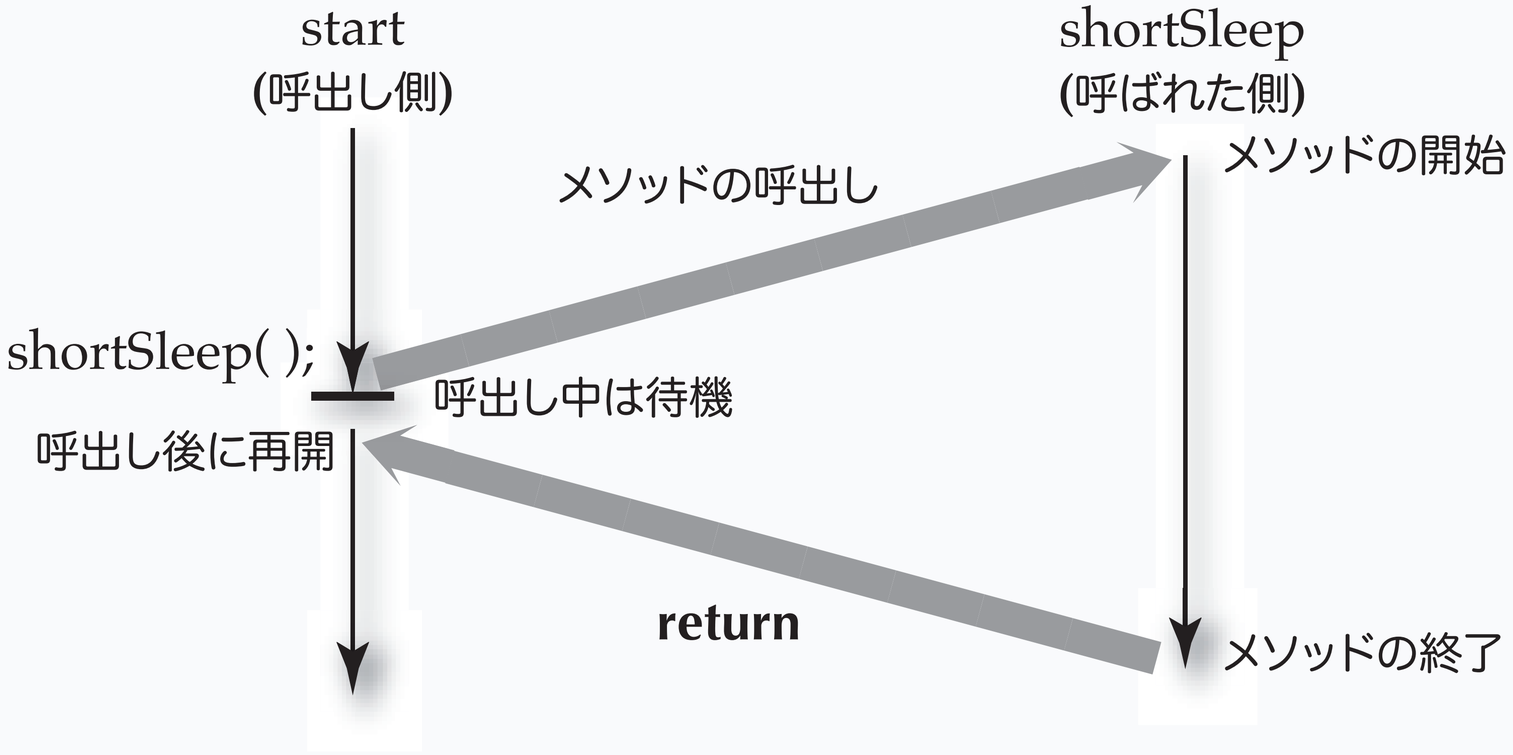
定義された関数であれば、関数名だけで呼び出せる

displayNumber( )

shortSleep( )

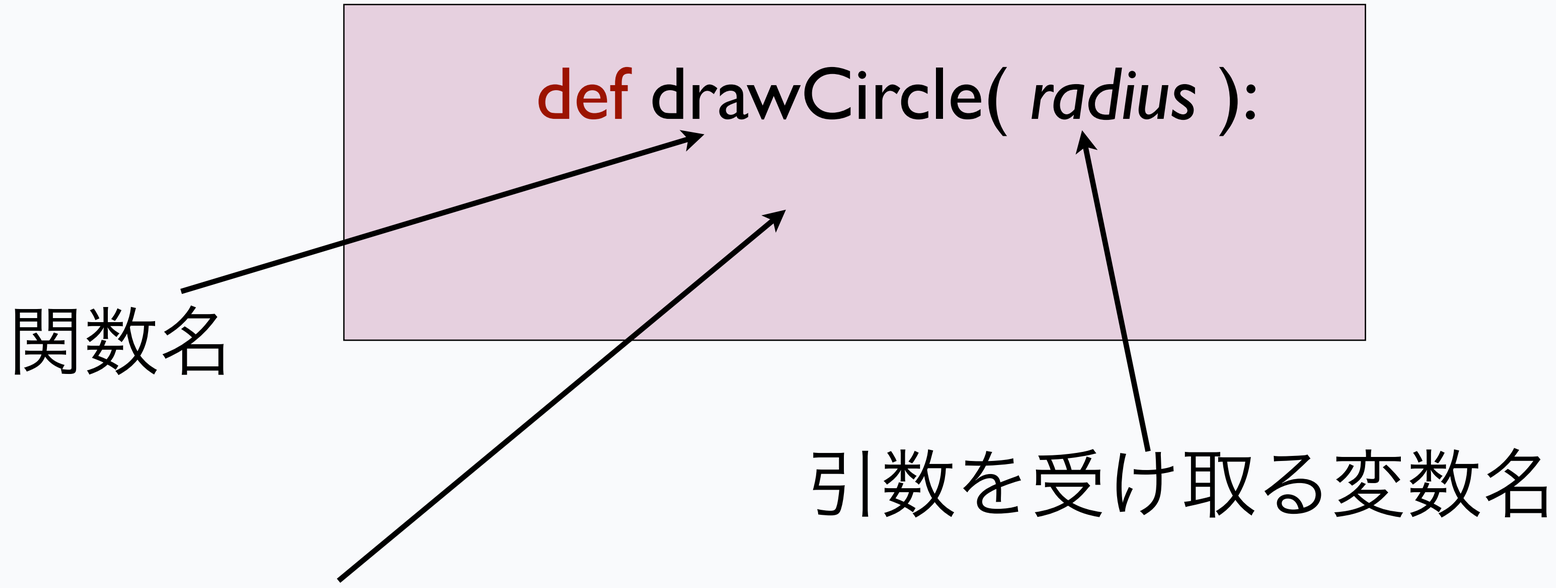
# 関数の呼出しの構造

- shortSleepの場合



# 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

- 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、まず評価されて、定数値（あるいはオブジェクトを指す値）になる式。実引数の値が、仮引数に代入されて、該当の関数が呼び出される。

# 引数のあるメソッドの例

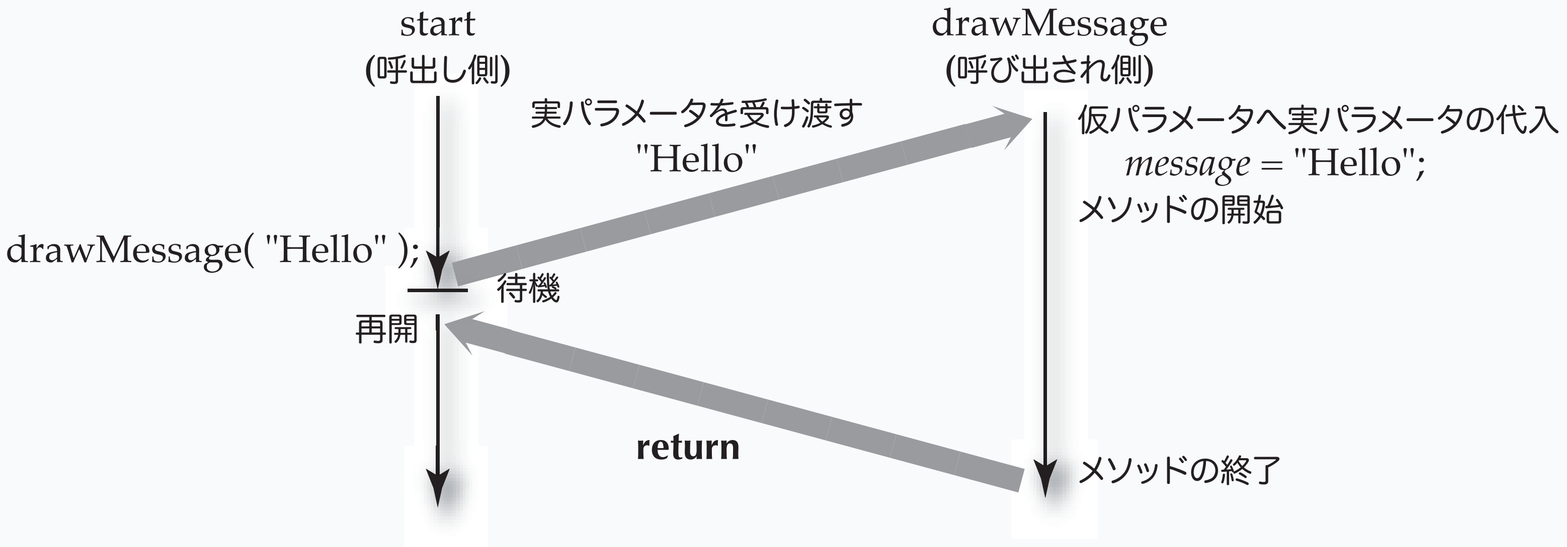
- 改行しないで表示を行なうdrawMessageの例

```
# 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）と呼ばれる

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

displayPower( 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（optional / keyword argument）
- 例：

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

- オプションの引数の場合は、オプション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( "Brian" )
displayPerson( gender="normal" )
```

# オプションの引数とキーワードの引数

- Pythonでは、オプションの引数（デフォルト値を持つ引数）とキーワード引数は、混在して用いることができる
- 例：
  - **def** sample( a, b, text="nothing" ): **pass**
  - 次の3つの形式で呼び出すことができる
    - sample( 12, 13, "something" )
    - sample( 12, 13, text="something" )
    - sample( 12, 13 )

# デフォルト値を持つ引数

- デフォルト値を持つ引数が定義されたら、それ以降、仮引数のところに*（アスタリスク）がでてくるまで、必ずデフォルト値を定義する必要がある
- 例：

```
def sample( a, b, c=10, d=20, *, e, f ): pass
def denied( a, b, c=10, d, e=20 ): pass
  # 許されない
def permitted( a, b, c=10, *, d, e=20 ): pass
  # これはOK
```

- 仮引数の「*」より後のものは、キーワード引数となり、必ず「キーワード変数=値」という形で呼び出す必要がある
- 例：

```
sample( 10, 20, 30, 40 ) # この呼出しはOK
sample( 10, 20, 30, 40, 50, 60 ) # これはだめ、e=50, f=60としなければならない
```

# 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
- 書式：
  - **def** 関数名( 位置引数, ..., /, オプション引数など ):
- 例：

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

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

f(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

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

- 例：

```
def g( a, /, b=20, *, c ): print( a, b, c )  # オプション引数のデフォルト値を定義したもの
g( 10, 20, c=30 )  # OK
g( 10, b=20, c=30 )    # OK
g( 10, c=30 )  # OK
```

# 可変の個数の引数

- 書式：

  **def** 関数名( *可変の個数を引き受ける変数名 ):

- 例：

  **def** printMessage( *message ): **pass**

- 関数内では、可変の個数を受け取った変数は、タプルとして利用することができる。また、「*変数名」の記法を使って、タプルを字面展開することが可能である。

- 例：

  **for** n **in** message: print( n ) # for文で使いたいとき

  print( message ) # そのままタプルとして渡される

  print( *message ) # 個々の要素が字面展開される

# 可変個数の引数とオプション引数を使う場合

- 先に可変個数の引数を定義しておく
- 例：

```
def displayPerson( name, *child,
gender="male" ):
  print( name, gender, end=" : " )
  print( ", ".join( child ) )


# 呼出し

displayPerson( "Bill Smith", "Tom", "Kent",
"Cathy" )
displayPerson( "Mary Brush", "Brian", "Jerry",
"Yoshua", gender="female" )
```

- 例：

```
def displayNumbers( *values, unit="" ):
    for i, value in enumerate( values ):
        print( value, end=unit )
        if i >= len( values )-1: print(); return
        print( end=", " )


# 呼出し

displayNumbers( 34, 45, 56 )
displayNumbers( 34, 45, 56, unit="meter" )
```

# キーと値がある可変の引数

- キーと値の対のキーワード引数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 ) # すべて字面展開される
def otherfunc( name="", age=0 ): print( name, age ) # 展開されたキーワード引数を持つ関数
printKeywords( name="John", age=23 )  # 呼び出してみる
```

# 位置引数と可変個のキーワード引数

- /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、その関数を呼び出す。
- ボトムアップの構造化（bottom up structuring)と呼ばれる

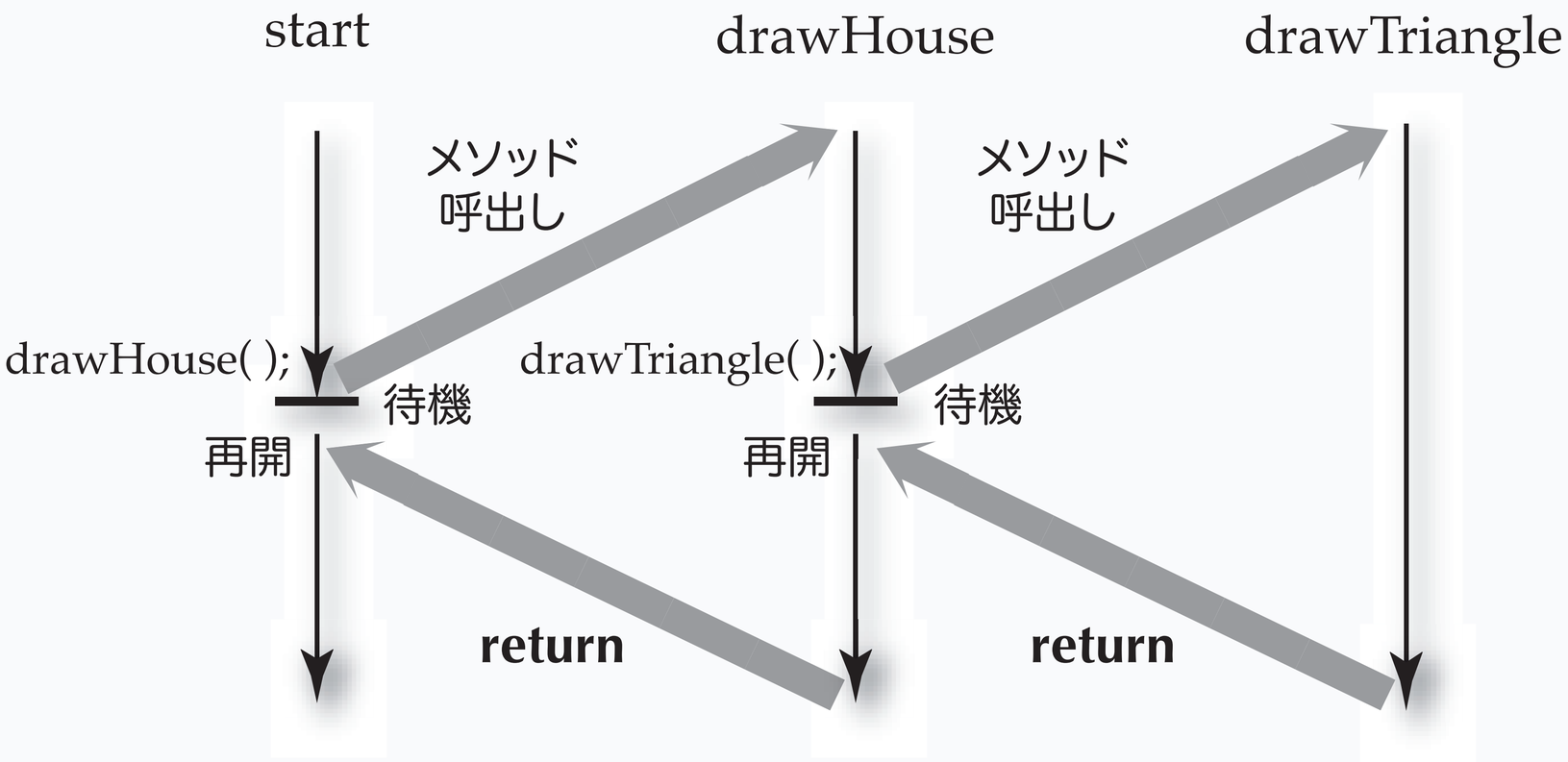
# 関数の構造化の意義

- 段階的詳細化（stepwise refinement）あるいは、トップダウン式の構造化（top down structuring )
  - 大きな作業を 1 つの関数として定義する。
  - その作業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

- 構造化プログラミング（structured programming）は、Edsger Wybe Dijkstra（エドガー・ダイクストラ）によって、1969年の方の論文で、段階的詳細化の手法として提案された。

# 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は、関数の実行と共に消える
- グローバル変数とローカル変数が同じ名前を使っていると、関数の中では、ローカル変数が優先される（グローバル変数を隠蔽する）
- 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 Variables

def sample( ):
    x=30 # Local Variables
```

# 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へのアクセス

- globals()関数
  - 現在の実行環境で定義されているグローバル変数の一覧を辞書（dict構造）で返す
  - globals()[ "変数名" ]でアクセスすることができる

- locals()関数
  - 現在の実行環境で定義されているローカル変数の一覧を辞書（dict構造）で返す

# 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 )
```

# 関数への引数としてのリスト

- 関数への引数としてのリストは、コピー渡しではなく、参照渡しで渡される
- 関数内で、引数のリストの内容を書き換えると、本体のリストも変更される
- 例：

```
def clear( nlist ):
    for i in range( len( nlist ) ):  nlist[ i ] = 0
numlist = [ 1, 2, 3, 4 ]
clear( numlist )
print( numlist )  # [0, 0, 0, 0]
```

- 関数で本体のリストを書き換えたくない場合、関数冒頭でcopyをする必要がある
- 例：

```
def noclear( nlist ):
    nlist = nlist.copy( )  # 新しいコピーされたリストで再設定
    for i in range( len( nlist ) ):  nlist[ i ] = 0
```

# 値を戻す関数

- **return**文を使う。そこで関数を終了し、式で記述された値を呼出し側に戻せる。
  - 関数の中で、
  - **return** 式
  - 例： **return** 34 * 32
- **return**の後に式がない場合は、ただ単に関数を終了させるだけ。
- 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 * x
```

- # ２つのうち、大きいほうの値を返す関数

```
# max( x, y )と同じ

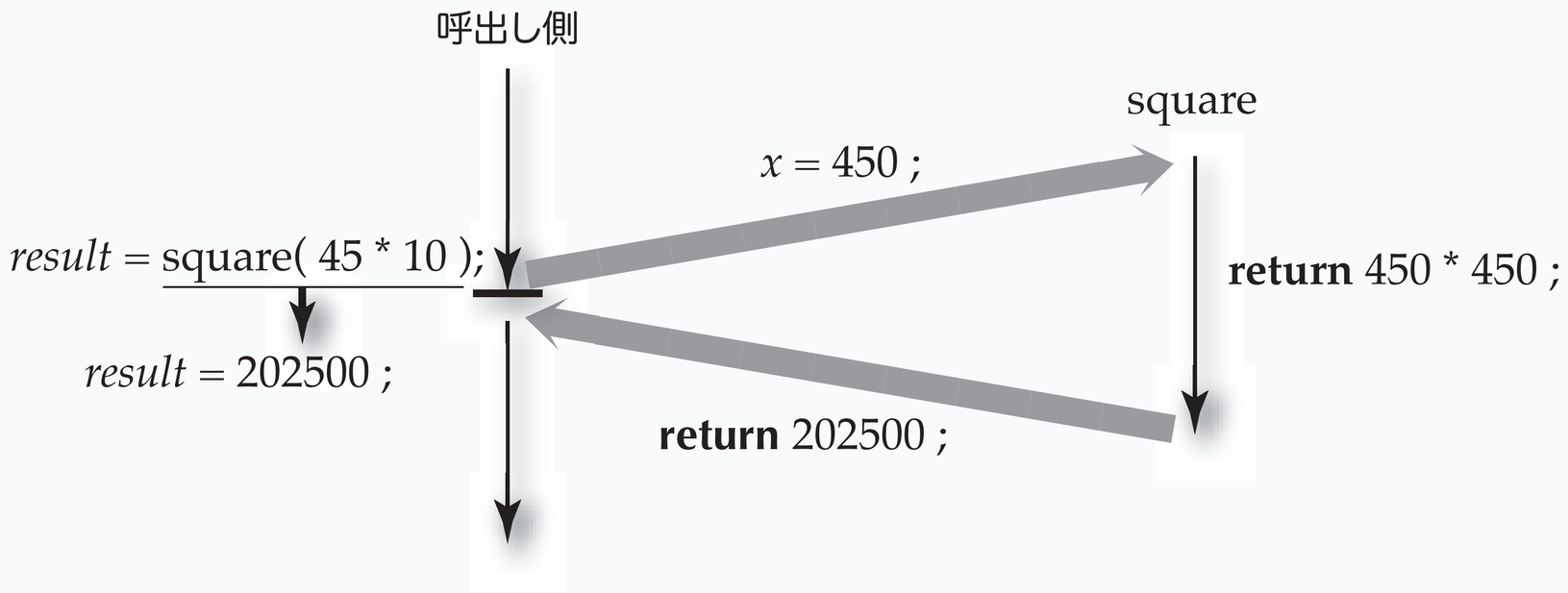
def  greater( x, y ):
     if  x > y : return  x
     else: return y
```

- 以下と同じ

```
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、呼び出された後に代入される。
  - 例： *result* = square( 45 * 10 )



# 値を戻す関数の多重呼出し

- 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）、sqaure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 ) # Python, Julia
- (more, lesser) = getMaxMin( 56, 89 )  # Python, CLU, JavaScript, Swift, Dart, C#, Julia

# 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、型ヒントが標準で導入された
- 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, ), 集合は{要素の型名}、辞書は[ キーの型名, 値の型名 ]で記述する
- 要素の型を指定しない場合は、リストは[]、集合は{}、タプルは(object,)、辞書は[object,]で記述する
- objectの替わりにanyも利用可能
- 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: **pass**
    **def** sample**(** i_dict: [ int, str ], n **:** int ) -> str: **pass**
    **def** sample( nlist: [], ntup: (object,) ) -> [int] : **pass**
    **def** sample( nlist: [any] ) -> any: **pass**

- 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

- 型を厳密にチェックしたい場合には、「mypy --strict ファイル名」で実行できる（インストールには、pip3/pip install mypyが必要）

# 約数と素数

- 約数を求めて、リストとして返す関数getDivisors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, …

# 素数の判定

- 試し割り法
  - 2から始めて、割れる数があれば、素数ではない
  - $n$が何らかの数pで割り切れる場合、$n$=pqであり、qがpより小さい場合には既にqもしくはqの約数で確かめた際に素因数が検出されているはずである。したがって、素因数候補として確かめるべきは、$\sqrt{n}$ までで充分となる。

    Wikipedia「素数」より。

```
for m in range( 2, int( n ** 0.5 ) + 1 ):
    if n % m == 0: return False
return True
```

# 完全数とピタゴラス数

- 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)

- 完全数とピタゴラス数を表示してみる

# 合同数

- 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）

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

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

- 自然数のピタゴラス数に対する整数の合同数を求めてみる

- 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。

# 完全数　求値の高速化の例

- ２進数から、完全数の候補を求め、その候補が完全数かどうかを判定する
- メルセンヌ数（$2^n$-1）が素数かどうか判定し、素数なら、$2^{n-1}$を掛ける。
- 素数かどうかを判定する関数に@jit修飾子を付けて、高速化して、時間計測をする

# numbaの高速関数

- Anacondaでは、標準的に用いることができる、spyder単体ではインストールできない（minicondaかcondaをインストールする必要がある）
- pipでインストールする際には、M1〜M4 Macの場合は、Intelのライブラリをアンインストールしてから、arch -arm64をつけて、llvmlite, scipy, numbaをインストーする
  - pip3 uninstall llvmlite, scipy
  - arch -arm64 pip3 install llvmlite, scipy, numba
- **from** numba **import** jit を記述する
- @jit修飾子を関数の定義の前に入れる
  - その関数は、実行前にコンパイルされるので、高速に動く

# timeモジュールのtime関数

- **import** time
- time.time( )で、世界協定時間（UTC： 1970年1月1日 00:00:00からの経過時間）が求まる、実数で秒単位
- 時間差を利用すれば、処理に掛かった時間を求めることができる
  - starttime = time.time( )
  - # 何らかの処理をここで行なう
  - elapsed = time.time( ) - starttime #経過時間を求める
  - print( elapsed, "秒経過" )

# 2進数（メルセンヌ素数）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

- 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

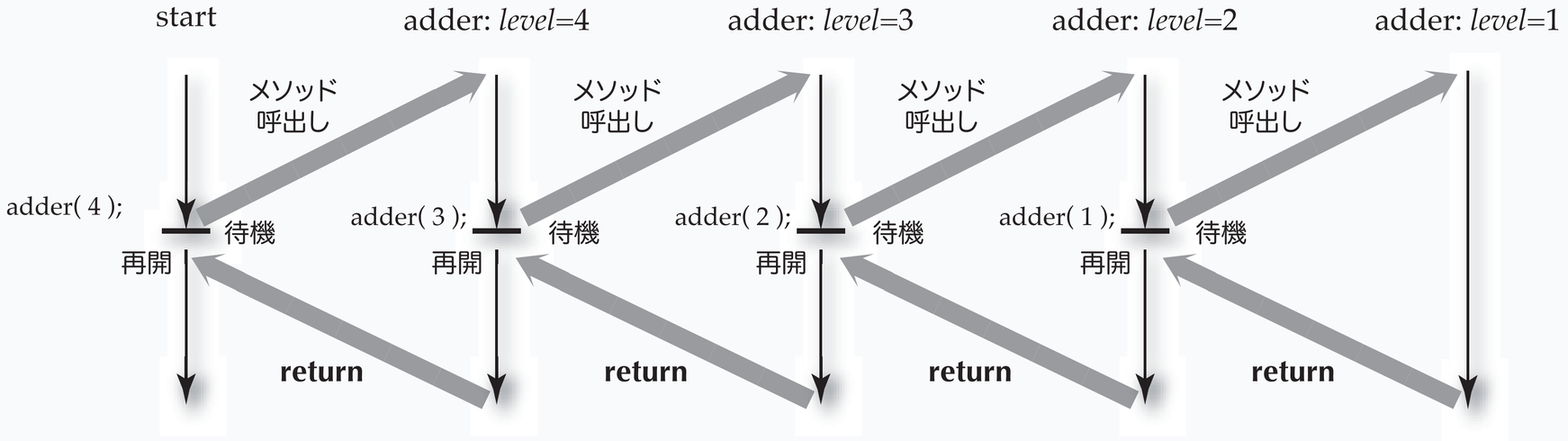
  ➡ ２．それ以上のレベルでの処理を記述

  　✳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
- 総和を計算するsummation
- $\sum_{i=0}^{n}$ 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

# 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
```

# 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

# 単位分数

- 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。

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

- このアルゴリズムの部分は、再帰を用いて記述する。
- [x]…xと等しいか、xよりも大きい整数の中で最小の整数（math.ceil）
- 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（high order function）

- 関数のパラメータとして関数を渡すもの。
- Javaだけはできないので、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 )
```

# 無名関数

- 高階関数を使うときに、パラメータを貰って値を返す関数を名前を定義しないで、指定することができる。
- 書式：

  **lambda** 仮引数: 返す値のための式
- 例：

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

  drawMessage( **lambda** n: "ABC" * n )
- 等価：
  - 下記の2つの式は、ほぼ等価になっている

    **lambda** パラメータ: 式

    **def** __無名関数__（パラメータ）: **return** 式

# リスト・タプル等とmap高階関数

- map組込み関数
  - 書式：map( 関数, リスト, .... )
  - リストの各要素に関数が適用されたシーケンスができる
  - 結果は、mapクラスのオブジェクトになっているので、tupleやlistに変換する
  - リストが複数ある場合、関数の引数は、リストの個数分だけ必要になる
- 例：
  - list( map( **lambda** n: n**2, range( 3, 10 ) ) ) ⇒ [9, 16, 25, 36, 49, 64, 81]
  - list( map( **lambda** a,b: a*b, range( 3, 6 ), [23, 34, 78] ) ) ⇒ [69, 136, 390]

# 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( "dummy:", x )
  def concrete( ): nonlocal x; x = 30; print( "concrete:", x )
  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
- 素数を作り出す関数を記述してみよう

# ジェネレータ式（generator expression）

- イテレータを返す式で、 普通の式に、ループ変数を定義する for 節、範囲、そして省略可能な if 節がつづいているようなもの。 こうして構成された式は、外側の関数に向けて値を生成する
  - 例：sum( n**2 **for** n **in** range(10) )
- ジェネレータ式の型は、generatorクラスのオブジェクトということになる
  - 例：type( n **for** n **in** range( 5 ) ) ⇒ <class 'generator'>

# 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
```

- 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 ) )
```

# 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 ) )

# 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 ) ) )

# デコレータの例（１）

- マイナス1をするデコレータを用いて、メルセンヌ数を求める例

```
# マイナス1をするデコレータ用の高階関数
def minus_one( f ): return lambda a: f(a)-1

# decoratorをつけて、関数を定義する
@minus_one
def mersen( p ): return pow( 2, p )

print( mersen( 10 ) ) # 1023
```

# デコレータの例（２）

- 結果を文字列型に変換して返すデコレータ関数の定義

```
def tostr( f ):
    def wrapper(*args, **kwargs):
        result = f(*args, **kwargs)  # 元の関数を実行
        return str(result)  # 結果を文字列に変換
    return wrapper
```

- デコレータ関数の使用例

```
@tostr
def square( n ): return n**2

print( [ square( n ) for n in range( 1, 11 ) ] )  # ['1',
'4', '9', '16', '25', '36', '49', '64', '81', '100']
```

- 関数のメタ情報（関数名やdocstring）を保持したい場合は、functools.wraps を使う
- 例：

```
from functools import wraps

def to_str(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return str(result)
    return wrapper
```