Python
Speed Learning

田浦健次朗
電子情報工学科

東京大学

Pythonの2つの実行方式

- ファイルに書いてファイルを実行

```
$ python ファイル名
```

- 対話的に実行

```
>>> 1 + 2
3
```

print 文

- 文法:

```
1 print 式
```

- 意味:
  - 「式」を計算 (評価) した値を表示する

print 文

- 例:

```
print 1
print 'hello'
print 1+2
print 7/3
print abs(-5*4)
```

print 文

- 例:

```
print 1
print 'hello'
print 1+2
print 7/3
print abs(-5*4)
```

- 出力:

```
1
hello
3
2
20
```

確認

- プログラムをファイルに保存して実行すると, 基本は, ファイルに並んだ文を上から実行していく
- 割り算 (/) の意味に注意
  - `7 / 3` ⇒ 2
  - `7 / 3.0` ⇒ 2.3333333333333335
  - `7 / float(3)` ⇒ 2.3333333333333335
  - 整数同士の割り算は余りを切り捨てる
  - どちらかを小数付きにすることで回避

変数への代入文

- 文法:

```
1 名前 = 式
```

- 効果:
  - 「式」の計算結果がその名前 (変数; variable) で記憶される
  - それ以降, 式中にその名前が現れたらその値はその計算結果になる

確認: 出力は?

- 以下をファイルに書いて, python で実行するとどんな出力になるか?

```
a = 1 + 2
print a
b = a + 3
print b
a = a + 4
print a
```

確認: 出力は?

- 以下をファイルに書いて, python で実行するとどんな出力になるか?

```
a = 1 + 2
print a
b = a + 3
print b
a = a + 4
print a
```

- 出力:

```
3
6
10
```

代入されていない変数を使うと...

- 例:

```
a = 1 + 2
print a
print b
```

代入されていない変数を使うと...

- 例:

```
a = 1 + 2
print a
print b
```

- 出力:

```
3
Traceback (most recent call last):
  File "a.py", line 3, in <module>
    print b
NameError:  name 'b' is not defined
```

- エラーメッセージをちゃんと見る事も重要

変数への代入文: 確認事項および補足

- 同じ変数に何度代入しても良い．変数の値はその都度変わる
- 「名前」は, アルファベット, アンダスコア (_), 数字の並び．ただし, 1 文字目が数字はダメ．
  - ◯: `x`, `y`, `tanaka`, `Kouji_Uehara`, `World_Cup_2013`
  - ×: `J-League`, `hungry?`, `4WD`

関数定義文

- 文法:

```
def 名前(名前, 名前, ...):
    文
    文
    ...
```

- 効果: 関数を「定義」する

関数定義文

- 例:

```
1  # 関数の定義
2  def f(x, y):
3      a = x + y
4      return a * a
5
6  # 定義した関数の使用 (呼び出し)
7  print f(10, 20) + 30
```

関数定義文

- 例:

```
# 関数の定義
def f(x, y):
    a = x + y
    return a * a

# 定義した関数の使用 (呼び出し)
print f(10, 20) + 30
```

- 出力:

```
930
```

確認

以下の中身のファイルを python で実行すると,

```
def f(x, y):
    a = x + y
    return a * a
print f(10, 20) + 30
```

1. まず f が定義される (中身は実行されない)
2. `print` 文が実行される.
   1. そのために,`f(10, 20) + 30` を計算.
   2. そのために, `f(10, 20)` を計算 ⇒ `x=10, y=20` として `f` の中身を実行する
3. 関数実行中に return 文:

```
return 式
```

に遭遇したら, 「式」の結果がその関数呼び出しの結果となり, 関数の実行はそこで終了する

意地悪クイズ: 出力は?

- 以下をファイルに書いて, python で実行するとどんな出力になるか?

```
def f(x, y):
    print 1
    print 'hello'
```

意地悪クイズ: 出力は?

- 以下をファイルに書いて, python で実行するとどんな出力になるか?

```
def f(x, y):
    print 1
    print 'hello'
```

- 出力

```
```

- 関数を「定義」しただけで中身が実行されるわけではない!

クイズ: 出力は?

```
def f(x, y):
    print x
    print y
    return x + y

print f(10, 20)
```

クイズ: 出力は?

```
def f(x, y):
    print x
    print y
    return x + y

print f(10, 20)
```

- 出力

```
10
20
30
```

クイズ: 出力は?

- 入力:

```
def f(x, y):
    print x
    print y
    x + y

print f(10, 20)
```

クイズ: 出力は?

- 入力:

```
def f(x, y):
    print x
    print y
    x + y

print f(10, 20)
```

- 出力:

```
10
20
None
```

クイズ: 出力は?

- 入力:

```
def f(x, y):
    print x
    print y
    x + y

print f(10, 20)
```

- 出力:

```
10
20
None
```

やりがちなミス (return の書き忘れ). 関数の中は以下のどちらかで終了する

- return 文を実行した時
- 上から順に文を実行し終え, 次に実行すべき文がなくなった時

後者の場合, その関数を呼び出した時の値は, None という特別な値となる

クイズ: 出力は?

- 入力:

```
1 def f(x, y):
2     print x
3     print y
4     x + y
5 
6 print f(10, 20) + 30
```

クイズ: 出力は?

- 入力:

```
1  def f(x, y):
2      print x
3      print y
4      x + y
5
6  print f(10, 20) + 30
```

- 出力:

```
1  10
2  20
3  Traceback (most recent call last):
4    File "a.py", line 6, in <module>
5  TypeError:  unsupported operand type(s) for +:
           'NoneType' and 'int'
```

クイズ: 出力は?

- 入力:

```
def f(x, y):
    print x
    print y
    x + y

print f(10, 20) + 30
```

None という値と, 30 を足し算しようとしたが, そんなことはできないというエラー

- 出力:

```
10
20
Traceback (most recent call last):
  File "a.py", line 6, in <module>
TypeError: unsupported operand type(s) for +:
        'NoneType' and 'int'
```

関数定義の中身が「どこまでか」は字下げで決まる

```
1 def f(x, y):
2     print 'hello'
3     print 'bye'
4 print 10
5 f(30, 40)
```

関数定義の中身が「どこまでか」は字下げで決まる

```
def f(x, y):
    print 'hello'
    print 'bye'
print 10
f(30, 40)
```

- 出力

```
10
hello
bye
```

- 関数定義の「中身」は print 'hello', print 'bye'
  - まず f が定義される (中身は実行されない)
  - print 10 は関数 f の定義の「外」 → 実行される
  - f(30,40) が実行され, 中身の print 'hello', print 'bye' が実行される

Python では字下げは文法の一部

- 気まぐれで字下げしてはいけない

```
def f(x):
    print x
      print x+1
```

```
  File "a.py", line 3
    print x+1
    ^
IndentationError:  unexpected indent
```

Python では字下げは文法の一部

- 字下げをやめるときはちゃんと「元に」戻る

```
print 10
def f(x):
    print x

 print 20
```

```
  File "a.py", line 5
    print 20
           ^
IndentationError:  unindent does not match any outer
        indentation level
```

字下げは Emacs におまかせ

- Emacs は Python のプログラム (xxx.py) を編集中, Tab キー (または C-i) を押すと, 適切な字下げをしてくれる
- Tab キーを何度も押すと, 「その場所で正当な字下げ」を次々に提案してくれる

関数がまた関数を呼べる

```
1 def f(x):
2     print x
3     y = x + 1
4     return y * y
5 def g(x):
6     y = f(x + 1)
7     print x
8     return x + y
9 print g(10)
```

関数がまた関数を呼べる

```
def f(x):
    print x
    y = x + 1
    return y * y
def g(x):
    y = f(x + 1)
    print x
    return x + y
print g(10)
```

- 出力

```
11
10
154
```

キーワード引数

- 通常関数を使用する際は, 値だけを並べる. 例:

```
def f(x, y):
  return x - y

print f(2, 3) # x=2, y=3
```

- 別の使い方として, x, y を明示的に渡すこともできる

```
print f(y=10, x=20)
```

- 関数に値を渡す際, 順番をいちいち気にしなくていい

キーワード引数

- 一方, 関数定義時には, 「渡されなかった場合の値 (デフォルト値)」を指定することができる

```
def g(x, y=1):
  return x - y

print g(2)   # x=2, y=1
print g(x=2) # x=2, y=1
print g(2, 3) # x=2, y=3
print g(x=2, y=3) # x=2, y=3
print g(y=3) # NG x は必須
```

- Visual Python, numpy などでは入力変数が多い高機能な関数が多く, 多用される

関数呼び出しが異なれば同じ名前であっても「別の変数」

- 先の例:

```
def f(x):
    print x
    y = x + 1
    return y * y
def g(x):
    y = f(x + 1)
    print x
    return x + y
print g(10)
```

において,

  - f 内の x (y) と g 内の x (y) は名前が同じでも,「別の変数」
- ややこしい? むしろ逆.
  - 関数実行中に代入された変数は「その」関数実行中だけのもの (局所変数)
  - 他の関数のことは考えなくて良い
  - 関数の入力変数 (引数) も同様

局所変数

```
def f(x):
    y = x + 1
f(10)
print y
```

- 出力:

```
Traceback (most recent call last):
  File "a.py", line 5, in <module>
    print y
NameError:  name 'y' is not defined
```

局所変数と大域変数

- ファイルのトップレベル (関数定義の外) で代入された変数は「大域変数」で, どの関数からでも参照できる

```
z = 10
def f(x):
    return x + z
print f(20)
z = 30
print f(20)
```

局所変数と大域変数

- ファイルのトップレベル (関数定義の外) で代入された変数は「大域変数」で, どの関数からでも参照できる

```
z = 10
def f(x):
    return x + z
print f(20)
z = 30
print f(20)
```

- 出力:

```
30
50
```

- だが, 大域変数 (とくに大域変数の書き換え) はむやみに使わないのが基本

import 文

- 文法:

```
from 名前 import *
```

- 効果:
  - 名前という名前の「モジュール」で提供されている拡張機能が使えるようになる

import 文の例

- 例:

```
print cos(0.1)
```

- 出力:

```
Traceback (most recent call last):
  File "a.py", line 1, in <module>
    print cos(0.1)
NameError: name 'cos' is not defined
```

- 例:

```
from math import *  # math モジュールを import
print cos(0.1)
```

- 出力:

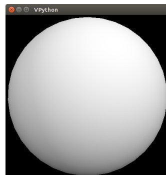
```
0.995004165278
```

import 文の例

- 例:

```
from visual import *
sphere()
```

- 出力:



import 文の変形

- 文法:

```
import 名前
```

- 効果:
  - `from 名前 import *` と似ているが, 個々の提供される名前は, 「`モジュール名.名前`」で提供される

```
import visual
visual.sphere()
```

- こちらの方が面倒だが, 複数のモジュールが同じ名前を定義していても OK なので, 「行儀の良い方法」とされる

Python のモジュール

- Python には無数のモジュールが提供されている
- Visual Python, Matplotlib, numpy などもすべて「モジュール」
- そもそもどんなモジュールがあるか?
  - ライブラリのマニュアルやインターネットで検索
- あるモジュールがどんな機能 (関数や変数) を提供しているか?
  - モジュールのマニュアル
  - python 中で, `dir(モジュール名)`
  -

```
from visual import *
print dir(visual)
```

```
['ALLOW_THREADS', 'BUFSIZE', ...
   ... 'sphere', ... ]
```

ここまでのまとめ

- print 文 (`print 式`)
- 代入文 (`名前 = 式`)
- 関数定義文 (`def` ...)
- return 文 (`return 式`)
- import 文 (`from 名前 import *`)
- 局所変数の概念

以降

より様々な,

- 文
- データの型 (数以外のデータ)

に触れる

for 文で繰り返し (1)

- 文法 (後に一般化):

```
1  for 名前 in range(式, 式):
2      文
3      文
4      ...
```

- 効果:

```
1  for i in range(a, b):
2      ...
3      ...
```

は, $i = a, a+1, \ldots, b-1$ に対して, 順に ... を実行

- 補足: `range(式)` は `range(0, 式)` の略記

for 文で繰り返し (1)：例 1

```
for x in range(3, 7):
    print x * x
```

for 文で繰り返し (1)：例 1

```
for x in range(3, 7):
    print x * x
```

- 出力

```
9
16
25
36
```

for 文で繰り返し (1)：例 2

```
1 def hatena(n):
2     a = 1
3     for i in range(0, n):
4         a = 2 * a + 1
5     return a
6 print hatena(5)
```

for 文で繰り返し (1)：例 2

```
def hatena(n):
    a = 1
    for i in range(0, n):
        a = 2 * a + 1
    return a
print hatena(5)
```

- 出力

```
63
```

for 文で繰り返し (1)：例 3

```python
def hatena(n):
    s = 0
    for i in range(0, n):
        s = s + i * i
    return s
print hatena(8)
```

- $0^2 + 1^2 + \cdots + (n-1)^2$ を求めよと言われて,

$$s = s + i * i$$

などという不気味な式を書けるようになることが，プログラミングに慣れる第一歩なのかも...

for 文で繰り返し (1)：例 3

```
def hatena(n):
    s = 0
    for i in range(0, n):
        s = s + i * i
    return s
print hatena(8)
```

- 出力

```
140
```

- $0^2 + 1^2 + \cdots + (n-1)^2$ を求めよと言われて，

$$s = s + i * i$$

などという不気味な式を書けるようになることが，プログラミングに慣れる第一歩なのかも. . .

if 文で場合分け

- 文法:

```
if 式1:
    文
    ...
elif: 式2:
    文
    ...
...
else:
    文
    ...
```

- elif 以降は省略可．elif は好きな数だけ, else: は最後にひとつ (または無し)
- 効果:
  - まず**式 1** を計算
  - 結果が偽 (0) でなければ最初の選択肢 (**文 1** ...) を実行
  - 偽だったら**式 2** を計算
  - 結果が偽 (0) でなければ 2 番目のの選択肢 (**文 2** ...) を実行

if 文で場合分け 例 (1)

```
def one_two_many(x):
    if x == 1:
        return ``ikko''
    elif x == 2:
        return ``niko''
    else:
        return ``takusan''
print one_two_many(2)
print one_two_many(3)
```

if 文で場合分け 例 (1)

```
def one_two_many(x):
    if x == 1:
        return ``ikko''
    elif x == 2:
        return ``niko''
    else:
        return ``takusan''
print one_two_many(2)
print one_two_many(3)
```

- 出力

```
niko
takusan
```

if 文で場合分け 例 (2)

```
def nabetsune():
    for i in range(1, 41):
        if i % 3 == 0 or i % 10 == 3 or 30 < i < 40:
            print i, "!!!!!"
        else:
            print i

nabetsune()
```

if 文で場合分け 例 (2)

```
def nabetsune():
    for i in range(1, 41):
        if i % 3 == 0 or i % 10 == 3 or 30 < i < 40:
            print i, "!!!!!"
        else:
            print i

nabetsune()
```

- 出力

```
1
2
3 !!!
...
```

より複雑なデータ

- これまで, 式の結果として得られる値は基本的には, ひとつの数値だけだったが
- 実用的なプログラム (e.g., ベクトル, 行列, 四角, 丸, 果物, 自動車 ...) のためには, より複雑な値, 特に「複数の値を組み合わせた値」が必要

- リスト
- タプル
- 文字列
- 辞書 (説明せず)
- オブジェクト

リスト

- 文法:

```
1  [ 式, 式, ..., 式 ]
```

- 意味: 各式を計算した結果を一本に束ねた値 (リスト)

リスト：例

- 例

```
from math import *
x = 20
a = [ 1, 1+2, cos(0.0), x ]
print a
print len(a)     # 要素数
print a[2]       # 特定の要素
b = a + [ 1.2, 3.4 ] # 2つのリストつなぐ
print b
c = [ a, a, a ] # リストのリスト
print c[1][2]    # c の 1 番目の要素(リスト)の 2番目の要素
```

- 出力

```
[1, 3, 1.0, 20]
4
1.0
[1, 3, 1.0, 20, 1.2, 3.4]
1.0
```

リスト：確認事項

- リストも数値と同様，「値」の一種に過ぎない
- 数値と同様，変数に代入したり，他の関数へ渡したりできる
- リストを受け取る関数，リストを返す関数，も書ける
- 「値の種類」が増えたこと以外，新しいことはない

```
def f(l):
  return l[0]

print f([1,2,3])
```

リストの書き換え

リストには, 値を追加・削除したり, 書き換えたりすることができる

```
a = [ 2, 3, 5 ]
a.append(7) # 末尾に追加
a[2] = 50
del a[1]     # a[1]を削除. 以降の要素は前にずれる
b = [ a, a, a ]
print b
a[2] = 30    # 注目
print b
```

```
[[2, 50, 7], [2, 50, 7], [2, 50, 7]]
[[2, 30, 7], [2, 30, 7], [2, 30, 7]]
```

リスト内包表記

- for 文で計算した結果を「一発で」リストにする強力な記法
- 覚えなくても問題ないが，覚えると賢くなった気がする
- 文法:

```
[ 式 for 名前 in range(式, 式) ]
```

- この式も, for 文同様本当はもっと一般的な文法 (後述).
- 意味: (あたかも「for 名前 in 式」を実行するように実行し，式を計算した結果をリストにする)
- 例を見たほうが早い

リスト内包表記：例

```
>>> [ x * x for x in range(0, 5) ]
 -> [0, 1, 4, 9, 16]
```

```
>>> sum([ x * x for x in range(0, 5) ])
 -> 30
```

それぞれ以下のようにしても同じことだがずっと簡潔

```
s = []
for x in range(0, 5):
  s.append(x * x)
```

```
s = []
for x in range(0, 5):
  s += x * x
```

(注: `s += ` E は `s = s + ` E と同じ意味)

タプル

- 文法:

```
1 式, 式, ..., 式
```

混乱しないよう, 習慣として括弧をつける

```
1 (式, 式, ..., 式)
```

- 意味: 各式を計算した結果を一本に束ねた値
- リストとほとんど同じ! 実際, 以下はリストと同様に使える
  - `len(...)` (要素数)
  - `...[...]` (指定要素)
  - `... + ...` (連結)

タプルとリストの違い

- タプルは, 一度作ったら要素の追加, 削除, 変更などはできない
-

```
a = (1.1, 2.2, 3.3)
a.append(4.4) # NG
del a[1]      # NG
a[1] = 22     # NG
a = (4.4, 5.5) # OK
a = a + (6.6, 7.7) # OK
```

- 最後の 2 つは, 変数を書き換えているのであってタプルを書き換えているのではない

タプルの典型的使用場面

- 二つ以上の値を一度に返す関数を手軽に書ける

```
def polar(x, y):
    r = sqrt(x * x + y * y)
    theta = atan2(y, x)
    return (r,theta)
```

- 変数に結果を受け取るときもこんなふうに書ける

```
r,theta = polor(3, 4)
```

- 実はこれでも「なぜリストだけじゃダメ?」の答えにはなっていないが
  - ちなみにこれも OK

```
[a0,a1,a2] = range(3,6)   # a0=3, a1=4, a2=5
```

文字列

- 文法:

```
"文字文字 ... 文字"
'文字文字 ... 文字'
"""文字文字 ... 文字"""
'''文字文字 ... 文字'''
```

- どれもほとんど同じ意味. なぜ色々ある?
  - 文字列中に"を含めたければ, ' が便利. 逆もまた然り
    - `a = 'he greeted, "hi"'`
    - `b = "Obama's lecture"`
  - 3 連打 (""", ''') は, 改行を含んでも良い
- そして, できる操作はまたしてもリストやタプルとそっくり
  - `len(...)`
  - `...[...]`
  - `... + ...`

文字列特有の操作: 値の埋め込み

- (例えば) ある変数 x の値を表示したい場合,

```
print x
```

- でもよいが, すぐに何がどこで表示されたのかわからなくなる.
もっとわかりやすく,

```
print "x = %s" % x
```

のようにする

- 例

```
from math import *
y = cos(3.14)
print "cos(3.14) = %s" % y
```

文字列への値の埋め込み：2 つ以上の値

- 2 つ以上の値を埋め込みたければ, %の右にタプルを書く
- 例

```
from math import *
x = 2.3
print "cos(%s) = %s" % (x, exp(x))
```

文字列への値の埋め込み：規則

- 一般に,

```
1 式 1 % 式 2
```

において, 式 1 の結果が文字列だったら, 上記の結果は式 1 中の左から i 番目の%s を, 式 2 の結果の第 i 番目の要素で置き換えた文字列

- %s 以外に, 表示したいデータの種類により, 色々あるがとりあえず %s は汎用的なのでこれを覚えれば良い
- 一応. . .
  - %d：整数
  - %9d：整数. ただし 9 文字以下は 9 文字分の幅になるよう右揃え
  - %f：浮動小数点数
  - %.3f：浮動小数点数. ただし, 小数点以下 3 桁まで
  - など

このゼミでのリスト・タプル・文字列の重要性 (1)

- Python 一般ににおいてはかなり重要な機能
- ベクトルや行列をつくろうと思ったら, 普通はこれらを使いこなすことになる
- が, このゼミではベクトルや行列は, もっとすごいの (visual の vector, numpy の多次元配列) を使うことを主眼にしているので, 深入りせずに先へ進む

より詳しい説明は Python チュートリアル 5 章「データ構造」を参照

このゼミでのリスト・タプル・文字列の重要性 (2)

- Python では, 実は異なるものが表面上同じ書き方で書け, 実際それが貫かれている, という設計思想 (オブジェクト指向, 多態性) に馴染むことも重要
  - リスト, タプル, 文字列どれにも, `+`, `len`, `[...]` などが適用可能
  - numpy の多次元配列や Visual Python の vector もそれらと似ている
- 全く同じではないところが時に混乱の元となる
- 「色々あるのだがそれらが似た表記で使えるように, 縁の下で頑張っている」という点を理解しておく

for 文で繰り返し (2)

- より一般化した文法:

```
for 名前 in 式:
    文
    文
    ...
```

- ここで, 式には, range(a, b) だけでなく, リスト, タプル, 文字列など, 色々なものが来れる
- 実は, range(a, b) は [a, $a+1$, $\ldots b-1$] というリストを作る関数に過ぎなかった

for 文で繰り返し (2)

- 例:

```
for x in [ 2, 3, 5 ]:
    print x
for x in "hello":
    print x
```

```
2
3
5
h
e
l
l
o
```

for 文で繰り返し (2)

- リストの各要素がタプルの場合はこんな書き方も許される

```
A = [ (0,1), (2,3), (4, 5) ]
for x,y in A:
    print x + y
```

- 出力

```
1
5
9
```

for 文で繰り返し (2)：zip

- zip(X, Y) という関数で二つの同じ長さのリストを，タプルのリストにできる
- 二つのリストからひとつずつ要素を取り出すような処理を for 文で書くのに重宝する

```
X = [ 1, 2, 3 ]
Y = [ 1, 4, 9 ]
for x,y in zip(X, Y):
    print x + y
```

- 出力

```
2
6
12
```

for 文で繰り返し (2)：enumerate

- `enumerate(L)` という関数で，リストの各要素を，そのインデクス (リスト内での位置; 0, 1, 2, . . .) とともに処理できる
- リスト内の出現位置を返したい場合や，出現位置が計算結果に意味を持つ場合に有用

```
def find_space():
  for i,c in enumerate("hello world"):
    if c == ' ':
      return i
  return -1
```

- 出力

```
5
```

その他の構文

- while 文：条件が成り立つ限り繰り返し (for 文より一般的な繰り返し)
- break 文：for, while を好きな時点で強制終了
- continue 文：for, while の「一回の繰り返し」を終了 (次の繰り返しへ進む)

グダグダ説明するのを省略して, Python チュートリアル第 4 章 「その他の制御フルーツール」を参照

オブジェクト

- オブジェクト：言葉通り「もの」
- Python においては, 色々な属性をひとまとめにした値
- たとえば,
  - 「球」は「中心」と「半径」
  - 「複素数」は「実部」と「虚部」
  - 「野球チーム」は「監督」と「選手のリスト」
  - etc.

クラス

- オブジェクトを生み出す「ひな形」「設計図」
- 文法:

```
class クラス名:
    関数定義
    関数定義
    ...
```

- 効果:
  - 「クラス名」で関数 (コンストラクタ) が定義され, 呼び出されると新しいオブジェクトが作られる
  - オブジェクトには属性値を代入できる

クラスとオブジェクト: 本当に最小の例

```
class nothing:
    pass  # 何もしない

# nothing クラスのオブジェクトを作る
m = nothing()
# 属性値へ代入 (≈ 変数への代入)
m.x = 10
m.y = 20
# 属性値を参照
print m.x + m.y
# もちろんオブジェクトを入力に取る関数も書ける

def take_nothing(n):
    n.x = n.x + 100

take_nothing(m)
print m.x
```

クラスとオブジェクト: もう少しそれらしい例

- __init__という関数で, コンストラクタの中身を定義できる
- その他好きな名前で関数 (メソッド) を定義できる
- メソッドの第一引数は作られたオブジェクトが渡される (普通 self という変数名を使う)
- ```
  class baseball_team:
      # コンストラクタが呼び出されるとこれが呼び出される
      def __init__(self, name, manager, players):
          self.name = name
          self.manager = manager
          self.players = players
      # メソッドの定義
      def add_player(self, p):
          self.players.append(p)
  ```
  ```

クラスとオブジェクト: もう少しそれらしい例

```
1  # オブジェクト作成 -> コンストラクタ__init__呼び出し
2  r = baseball_team("Nipponham", "Kuriyama",
3                    [ "Inaba", "Saito", "Takeda" ])
4  print r.players
5  # メソッド呼び出し
6  r.add_player("Ohtani")
7  print r.players
```

クラスとオブジェクト: もう少しそれらしい例

```
# オブジェクト作成 -> コンストラクタ__init__呼び出し
r = baseball_team("Nipponham", "Kuriyama",
                  [ "Inaba", "Saito", "Takeda" ])
print r.players
# メソッド呼び出し
r.add_player("Ohtani")
print r.players
```

```
[ "Inaba", "Saito", "Takeda" ]
[ "Inaba", "Saito", "Takeda", "Ohtani" ]
```

このゼミと, クラス・オブジェクト

- 大きなプログラムを作るのにクラス・オブジェクトは非常に重要
- このゼミでは, 自らクラス・オブジェクトを作る必要はあまりない
- 一方, Visual Python, numpy を始め, 提供される機能は大部分が, オブジェクトとして提供されるため, 「どんな風に使うか」はマスターする必要あり (メソッドを呼ぶ, 属性値を参照・変更する)
- 以下の例でもオブジェクトをいくつも作っている (`vector`, `sphere`, `+`いずれもオブジェクトを返す)

```
center = vector(2, 3, 4)
s = sphere(pos=center)
s.pos = center + vector(1, 1, 1)
```

特別なメソッド

- 例えば`__add__(self, o)`というメソッドを定義すると, 足し算 (+) を定義できる
- `x + y`は, `x.__add__(y)`の意味

```
class vector:
  def __init__(self, x, y, z):
    self.x = x
    self.y = y
    self.z = z
  def __add__(self, o):
    return vector(self.x + o.x, self.y + o.y,
                  self.z + o.z)

u = vector(1,2,3)
v = u + vector(4, 5, 6) # u.__add__(vector(4, 5, 6))
print v
```

- 他にも, `__sub__`, `__mul__`, `__div__`など

特別なメソッド

- Python ではこうして, +, -, *など, 馴染みの記法で, オブジェクトに相応しい動作を行わせることができる
- Python のリスト, タプル (連結), Visual Python の vector (ベクトルの和), numpy の array, mat など
- それらの動作は通常, 自然なためあまり意識することは無いが, 決して人間のように「文脈から」「空気から」判断しているわけではないので注意
- 例えば, vector とリストをごっちゃにすると, +の動作の違いであたふたすることになる

Visual Python プログラム例を改めて鑑賞

```
from visual import *
s1 = sphere(color=color.red) # キーワード引数
s2 = sphere(pos=vector(1.0, 2.0, 3.0), radius=0.1)
# s1, s2 はオブジェクト
rate(0.3) # 約3.3秒待つ (0.3フレーム/秒)
# 属性へ代入
s2.color = color.green  # 色が緑に変わる
rate(0.3) # 約3.3秒待つ (0.3フレーム/秒)
s2.pos = s2.pos + 0.5 * vector(-1.0,-1.0,-1.0) # 球が「動く」
```

- Visual Python の sphere などの関数は, キーワード引数を多数受け取る
- 作られた物体の属性 (color, radius, pos など) に代入すると, その物体の見た目が変わる
- 実は, 自分の都合で, 自由な名前で属性を代入したりして OK (例えば速度が必要なら, `s1.vel =` とか)

Visual Python プログラム例を改めて鑑賞

```
1 from visual import *
2 s1 = sphere(color=color.red) # キーワード引数
3 s2 = sphere(pos=vector(1.0, 2.0, 3.0), radius=0.1)
4 # s1, s2 はオブジェクト
5 rate(0.3) # 約 3.3秒待つ (0.3フレーム/秒)
6 # 属性へ代入
7 s2.color = color.green  # 色が緑に変わる
8 rate(0.3) # 約 3.3秒待つ (0.3フレーム/秒)
9 s2.pos = s2.pos + 0.5 * vector(-1.0,-1.0,-1.0) # 球が「動く」
```

- Visual Python の vector で 3 次元ベクトルを作れる
- + はベクトルの足し算, *はスカラー倍という自然な動作をする

Visual Python の vector

```
>>> from visual import *
>>> v = vector(1,2,3)
>>> v
vector(1, 2, 3)
>>> v[0]
1.0
>>> v.x
1.0
>>> 2 * v
vector(2, 4, 6)
>>> v + v
vector(2, 4, 6)
>>> dir(v)
[ ..., 'astuple', 'clear', 'comp', 'cross', 'diff_angle', 'dot', 'mag', 'mag2', 'norm', 'proj', 'rotate', 'x', 'y', 'z']
```

Visual Python の vector

```
1  >>> v.mag
2  3.7416573867739413
3  >>> v.mag2
4  14.0
5  >>> v.dot(v) # 内積
6  14.0
7  >>> v.norm()
8  vector(0.267261241912424, 0.534522483824849,
           0.801783725737273)
9  >>> v.mag * v.norm()
10 vector(1, 2, 3)
11 >>> v.cross(vector(1,1,1)) # 外積
12 vector(-1, 2, -1)
```

Visual Python プログラム例を改めて鑑賞

```
from visual import *
s1 = sphere(color=color.red) # キーワード引数
s2 = sphere(pos=vector(1.0, 2.0, 3.0), radius=0.1)
# s1, s2 はオブジェクト
rate(0.3) # 約 3.3秒待つ (0.3フレーム/秒)
# 属性へ代入
s2.color = color.green  # 色が緑に変わる
rate(0.3) # 約 3.3秒待つ (0.3フレーム/秒)
s2.pos = s2.pos + 0.5 * vector(-1.0,-1.0,-1.0) # 球が「動く」
```

- Visual Python の `rate(`f`)` 関数は, 画面の更新と, f フレーム/秒になるような時間調節 ($\approx 1/f$ 秒休む)
- rate を呼ばないと画面の更新がないまま計算が進行し, 最終状態だけが表示される (アニメーションにならない) ので注意

プログラム中のコメント

- プログラム中に注釈 (コメント) を書くことは, 実はすごく重要
- Python では, 各行#以降がコメント

```
1  def main():
2      # location of the pointmass
3      p = vector(1,2,3)
4      # the pointmass
5      s = sphere(pos=p)
6      # the spring
7      sp = helix(pos=p)
8      ...
```

- 日本語で書きたい場合, ファイルの 1 行目に以下を書いておく

```
1  #!  -*- coding:  utf-8 -*-
```