Python
Speed Learning

田浦健次朗
電子情報工学科

東京大学

Contents

プログラミング言語の習得

- 道具・概念を覚える
  - どのような場面でどの道具が必要になるか
  - 関数 (def)，変数，繰り返し (for)，条件分岐 (if)，リスト，配列，etc.
  - ≈ 数学でどの問題にどの公式を使うか
  - 言語が違っても共通部分大きい
- 文法規則を理解する
  - 1 文字でも間違ったら許してくれない
  - 文法規則の表記法を理解し,「許されるプログラム」がわかるようになる
- 実行規則を理解する
  - 「このプログラムを実行したら何が起きるのか」をしっかり理解する
  - エラーがどうして起きたのか理解し，対処できるようになる

Jupyter 環境について最低限の事

- 入力セルには，Python の「式」または「文」(両者の区別は後ほど) を入力する
- Shift + Enter でそのセルの中身を「実行」する．それが「式」だったら結果が表示される

```
In [20]: 1 + 2

Out[20]: 3
```

- 一つのセルに複数の式 (文) を書いてもよい．結果が表示されるのは最後の式 (文) だけ (実行は全て，されている)．

```
In [21]: 1 + 2
         3 + 4

Out[21]: 7
```

- 最初は，セルに 1 行入れてはその結果を見る，という使い方をするが，徐々にひとつのセルに大きなプログラムを書いていくようになる

このスライドでの入力とその結果の表記

- このような実行例:

```
In [20]: 1 + 2
Out[20]: 3
```

を，このスライドでは

```
1 + 2
→ 3
```

のように表記

Contents

式: 足し算，掛け算，累乗

```
1 + 2
→ 3
```

```
3 * 4
→ 12
```

```
5 ** 6                          # = 5^6
→ 15625
```

```
7890 ** 1234                    # = 7890^1234 巨大な数も軽々
→ 9840608642752423747765046213370244799217802266 36...
```

その他の式

- 余り

```
13 % 7
→ 6
```

- 等号は=じゃなくて==

```
1 == 2
→ False
```

- 大小比較

```
3 < 4
→ True
```

```
5 >= 6
→ False
```

- かつ，または

```
7 <= 8 and 9 == 9.0
→ True
```

```
7 > 8 or 9 != 9.0
→ False
```

- 数学で慣れた記法

```
10 < 11 < 12
→ True
```

- 実は True とは 1 (False とは 0) のこと

```
True + 3
→ 4
```

Contents

関数

いくつかの関数が Python に組み込まれている (一覧: https://docs.python.jp/3.5/library/functions.html)

```
1 abs(-2)
2 → 2
```

```
1 float(3)
2 → 3.0
```

```
1 int(4.5)
2 → 4
```

```
1 min(10,20,30,40)
2 → 10
```

```
1 (abs(-2) + float(3)) / max(50,60,70,80,90)
2 → 0.0555555555555555
```

print 関数

- 効果: 実行されると，⟨ 式 ⟩ を評価した結果をその場で表示する
- 例:

```
1 print(1 + 2)
2 print(7 / 3)
3 print(10 < 11 < 12)
4 → 3
5 2.3333333333333335
6 True
```

- 実行結果や途中経過を表示する道具

Contents

追加機能の輸入 (import)

- 多くの関数は初めから組み込まれていない

```
1 sin(1.0)
2 → (most recent call last):
3  File "<stdin>", line 1, in <module>
4 name 'sin' is not defined
```

- 所望の機能 (モジュール) を「使いたい (import)」と宣言して初めて使える.

```
1 import math      # math モジュールを import
```

```
1 math.sin(1.0)    # 「モジュール名. 関数名」で目当ての関数が使える
2 → 0.8414709848078965
```

math モジュールの頻出機能

- sin, cos, tan, π, e, exp, log, ...

```
math.sin(1.5)
→ 0.997494986604054
```

```
math.cos(1.5)
→ 0.0707372016677029
```

```
math.pi
→ 3.141592653589793
```

```
math.sin(math.pi)
→ 1.2246467991473532e-16
```

```
math.e
→ 2.718281828459045
```

```
math.exp(1.0)
→ 2.718281828459045
```

```
math.log(math.e)
→ 1.0
```

```
math.log(81, 3)
→ 4.0
```

モジュールとの付き合い方

- 文字通り無数の機能が「モジュール」として提供されている
- 一部は Python に常に付属している標準モジュール，一部は追加インストールが必要な外部モジュール
- 本ゼミで使う，visual, matplotlib, numpy, scipy などもすべて「(外部) モジュール」
- 暗記は不可能．必要なモジュールを探し，その機能を調べられることが大事
  - Google 検索: python math モジュール とか
  - Python の標準ライブラリ一覧: `https://docs.python.jp/3.5/library/`

Python 環境内でモジュール・関数の機能を調べる方法

- dir: モジュール内で定義されている名前の一覧

```
1 dir(math)
2 → [ ...  'cos', 'cosh', ....  ]
```

- help:

```
1 help(math.sin)
2 → Help on built-in function sin in module math:
3
4     sin(x)
5
6     Return the sine of x (measured in radians).
```

```
1 help(math)
2 → math モジュールに関する長〜い説明
```

importの色々なスタイル

- 正当:

```
1 import math
2 math.sin(1.0)
```

- 略記を定義:

```
1 import numpy as np
2 np.array([1,2,3])
```

- モジュール名を省略 + 輸入する名前を限定

```
1 from math import sin,cos,log
2 log(sin(1.0)**2 + cos(1.0)**2)
```

- モジュール名を省略 + 全部の名前を輸入

```
1 from visual import *
2 vector(1,2,3)
```

Contents

変数

- これまで Python で計算できる色々な式の形 (四則や関数) を見た
- ここまではいわば関数電卓レベル
- プログラミング言語には，「変数」というものがあり，計算の途中結果に名前を付けて (代入文)，後の計算に利用できる
- 例:

```
c = math.cos(1.0)    # 変数 c に結果を格納 (代入)
```

```
c
→ 0.5403023058681398
```

```
s = math.sin(1.0)
```

```
c ** 2 + s ** 2      # 式に変数を使える
→ 1.0
```

変数の使いみち

≈ 「値」に名前をつける

- 式を見やすくする

```
3.1415926535 * 2.589 * 2.589
```

と書く代わりに

```
pi = 3.1415926535
r = 2.589
pi * r * r
```

- 同じ式を 2 度書かなくてすむようにする
- (主に for 文と組み合わせて) 変数の値を書き換えながら長い計算をする

同じ変数に何度も代入できる

- 同じ変数に何度代入しても良い．変数の値は「最後に代入された値」

```
1  x = 3
2  x = 4
3  x
4  → 4
```

- 代入の右辺にも変数を使って良い．`x = x + 1` などという，数学で見慣れない式に慣れるのも，プログラミングの一部

```
1  x = 4
2  x = x + 1    # x = 4 + 1
3  x
4  → 5
```

Contents

関数定義

- 自分で新しい関数を定義し，使うことができる → プログラミングで最重要な概念
- 例:

```
1 def f(x, y):                    # 関数 f を定義
2     return x ** 2 + y ** 2
```

```
1 f(3, 4)                         # x=3, y=4 として関数 f を使う（呼び出す）
2 → 25
```

関数本体に複数の文を並べても良い

- 使い道: 長い式を一気に書くのではなく，変数などを使って適宜分解し，見やすくする
- 例: a, b, c から，

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

を計算する関数 `solve_q`:

- 書き方 1:

```
1 def solve_q(a, b, c):
2     return (-b + math.sqrt(b*b - 4*a*c)) / (2*a)
```

- 書き方 2:

```
1 def solve_q(a, b, c):
2     D = b * b - 4 * a * c
3     return (-b + math.sqrt(D)) / (2 * a)
```

関数 (定義) の使いみち

≈ 「計算の過程」に名前をつける

- 式を見やすくする

```
root = (-b + math.sqrt(b*b - 4*a*c)) / (2*a)
```

の変わりに

```
root = solve_q(a, b, c) # 2次方程式を解く
```

- 同じ文・式・その組み合わせを 2 度書かなくてすむようにする

「変数」と「関数」を組み合わせて, 意味がわかりやすく, 値の変更がしやすいプログラム

プログラミング≈関数の定義，関数の定義，...

- 難しい問題を解くプログラムは，多数の関数から成る
- 簡単な関数を元に，それを部品として使って別の関数を作り，...
- 徐々に複雑な計算を組み立てていく

関数呼び出し実行の規則

- 関数呼び出し (e.g., `f(a)`) が実行されたら，`f` の本体の文が上から順に実行される
- その最中 “`return` E” が実行されたら E の結果が `f(a)` の結果になる．`f` の中にそれ以降文があっても実行されない．

```
1 def f(x):
2     return x + 1
3     return x + 2  # 実行されない
```

```
1 f(3)
2 → 4
```

関数定義の落とし穴

- 注: return が実行されることなく最後の文が終了すると，None という謎な値になる

```
1 def f(x):
2     x + 1              # たぶんreturn 文を書き忘れた
```

```
1 f(4)     # 何も表示されない (実はNone が「表示」されている)
```

```
1 f(4) + 2              # None + 2 を実行することになり,エラー
2 → Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError:  unsupported operand type(s) for +:
          'NoneType' and 'int'
```

プログラムは「文法」に沿って書かなくてはならない

- 式, 変数, 関数呼び出し, ... などを習っていったとき, それらを正しく組み合わせられるよう, すべてがひとつの「文法」にしたがっているということを意識するとよい
- 英語の文型 文 = S V O | S V O C | ... みたいなものだが, 英語と異なり「融通もないが例外もない」→ 意識して覚える必要・価値が高い

ここまでの文法 (1)

- セルに書いて良いもの

```
1 ⟨ セル ⟩ ::=
2 ⟨ 文 ⟩
3   ...
4 ⟨ 文 ⟩
```

- (これまでに出てきた) 文:

```
1 ⟨ 文 ⟩ ::=
2  return ⟨ 式 ⟩      # return 文
3 | x = ⟨ 式 ⟩        # 代入文
4 | import m          # import 文
5 | ⟨ 関数定義 ⟩      # 実は関数定義も文
6 | ⟨ 式 ⟩            # 実は式も文の一種
```

関数定義の文法

- 関数定義の本体には,「文」を任意個並べることができる

```
1 ⟨ 関数定義 ⟩ ::=
2 def f(x, y, ···):
3     ⟨ 文 ⟩
4       ···
5     ⟨ 文 ⟩
```

式の文法

- (これまでに出てきた) 式

```
1  ⟨ 式 ⟩ ::=
2    数                                  # 1とか 2.3とか
3  | x                                   # 変数の参照
4  | ⟨ 式 ⟩ ⟨ 演算子 ⟩ ⟨ 式 ⟩            # 3 + 4とか
5  | ⟨ 演算子 ⟩ ⟨ 式 ⟩                   # -3 とか
6  | ⟨ 式 ⟩ (⟨ 式 ⟩, ⟨ 式 ⟩, ···)        # abs(-3)とか
7  | ⟨ 式 ⟩ . x                          # math.sin とか
8  | ( ⟨ 式 ⟩ )
```

- 上記で f, x, m は変数名，関数名，モジュール名を表し，文法上は「識別子」と呼ばれる．
- 識別子として許されるのは，アルファベット (`A-Z`, `a-z`)，下線 (`_`) および数字 (`0-9`) の並び．ただし先頭に数字は許されない．
  - OK: `x`, `y`, `sin`, `my_func`, `Masahiro_Tanaka_19`
  - NG: `x'`, `1st_attempt`, `Masahiro Tanaka 19`

文法の落とし穴 (1)

特に，他のプログラミング言語を知っている人は注意

- 式が長くても勝手に改行してはいけない

```
1  def f(x):
2      return 3 * x * x * x * x
3           + 4 * x * x * x
4  → File "<...>", line 3
5      + 4 * x * x * x
6      ^
7  IndentationError:  unexpected indent
```

- def f(x) のあとの，コロン (:) を忘れない!

```
1  def f(x)
2      return x ** 2
3
4  → File "<...>", line 1
5      def f(x)
6              ^
7  SyntaxError:  invalid syntax
```

文法の落とし穴 (2)

- 関数本体は必ず字下げをする; 字下げされているところが関数内部

```
1 def f(x):
2     y = x + 1
3     return y * y
4     f(10)
```

f(10) を実行しているつもりかも知れませんが，実行されません

- 字下げは揃える (字下げすればいいわけではない)

```
1 def f(x):
2     y = x + 1
3       z = y * y
4   return z ** 2
5 → File "<...>", line 3
6     z = y * y
7     ^
8 IndentationError:  unexpected indent
```

- 今後も何かとこのパターン (コロン (:) のあと字下げ) が出てきます

Contents

さらなる文

- これまでに学んだ文 (主に代入文, return 文) と式 (算術式と組み込み関数) だけで書ける関数はたかが知れている
- for 文と if 文を習うことで，かなりのことが書けるようになる

```
1 ⟨ 文 ⟩ ::=
2   return ⟨ 式 ⟩      # return 文
3 | x = ⟨ 式 ⟩         # 代入文
4 | import m           # import 文
5 | ⟨ 関数定義 ⟩       # 実は関数定義も文
6 | ⟨ 式 ⟩             # 実は式も文の一種
7 | for 文
8 | if 文
```

繰り返し: for 文の初歩

- 文法: 本当はもう少し一般的だがまずは以下の形で:

```
for x in range(E0,E1):
    ⟨ 文 ⟩
     ···
    ⟨ 文 ⟩
```

- E_0, E_1 は任意の ⟨ 式 ⟩
- 実行規則 ($\approx$ 本体を $E_1 - E_0$ 回繰り返す):
  1. E_0, E_1 を計算 (それぞれ a, b とする)
  2. $x = a$ として ⟨ 文 ⟩ $\cdots$ ⟨ 文 ⟩ を実行
  3. $x = a + 1$ として ⟨ 文 ⟩ $\cdots$ ⟨ 文 ⟩ を実行
  4. $\cdots$
  5. $x = b - 1$ として ⟨ 文 ⟩ $\cdots$ ⟨ 文 ⟩ を実行
- E_1 を省略可. `range(`E`)` $\equiv$ `range(0,` E`)` ($\approx$ E 回繰り返し)

(あまり役に立たない) for 文の例

- 0 から $n-1$ まで表示

```
1  def print_i(n):
2      for i in range(n):
3          print(i)
4
5  print_i(10)
6  → 0
7  1
8  .
9  .
10 9
```

- 注:
  - `print(..)` 文は実行の途中経過を知る良い手段
  - ただし大量に `print` すると Jupyter 環境で

for 文の中の for 文

- for 文自体が ⟨ 文 ⟩ の一種で, for 文の中には任意の ⟨ 文 ⟩ が書けるのだから以下は正しい文

```
1  def print_ij(n):
2      for i in range(0, n):
3          for j in range(10, 10 + n):
4              print(i, j)
5
6  print_ij(3)
7  → 0 10
8  0 11
9  0 12
10 1 10
11 1 11
12 1 12
13 2 10
14 2 11
15 2 12
```

for 文の例

- n が与えられ，1 から n までの和:

$$\sum_{k=1}^{n} k^2 = 1^2 + \cdots + n^2$$

を計算する

```
1  def sum_k2(n):
2      s = 0
3      for k in range(1, n+1):
4          s = s + k * k
5      return s
6
7  sum_k2(10)
8  → 385
```

for 文の実行を理解する

```
1 def sum_k2(n):
2     s = 0
3     for k in range(1, n+1):
4         s = s + k * k
5     return s
```

- sum_k(10) 実行の様子:

```
1 s = 0
2 s = s + 1 * 1     # s = 1
3 s = s + 2 * 2     # s = 5
4 s = s + 3 * 3     # s = 14
5    ...
6 s = s + 10 * 10   # s = 385
7 return s
```

for 文のあるある間違い

```
1  def sum_k2(n):
2      for k in range(1, n+1):
3          s = s + k * k
4      return s
5
6  sum_k2(3)
7  →
```

```
1  ---------------------------------------------------------------------------
2  UnboundLocalError                         Traceback (most recent call last)
3  <ipython-input-6-bb13f487ff09> in <module>()
4  ----> 1 sum_k2(3)
5
6  <ipython-input-5-0e7b0b967ad8> in sum_k2(n)
7        1 def sum_k2(n):
8        2     for k in range(1, n+1):
9  ----> 3 s = s + k * k
10       4     return s
11
12 UnboundLocalError:  local variable 's' referenced before assignment
```

このエラーの起きた理由

1. sum_k2(3) を実行しようとした
2. →

```
for k in range(1, 4):
    s = s + k * k
```

3. →

```
    s = s + 1 * 1
```

4. → しかし (右辺の)s はまだ一度も代入されていない (値が不定) ためエラー

```
    s = s + 1 * 1
```

5. 以下と同種のエラー

```
def f(x):
    return y

f(3)
```

教訓 (エラーとの付き合い方)

- プログラムの実行規則はいつも同じ. 普通に実行していって, ありえない (これ以上進めない) 状況に遭遇するとエラーになる
- エラーの記録から以下を読み取る
  - エラーの種類・説明 (最後の行)

```
UnboundLocalError: local variable 's' referenced before assignment
(局所変数's'が代入される前に参照された)
```

  - どこでエラーが起きているか (最後の----->)

```
----> 3          s = s + k * k
```

  - そこに至るまでの関数呼び出しの履歴

```
----> 1 sum_k2(3)
```

- ⇒ `sum_k2(3)` を実行中, その中の for 文を実行中, その中の `s = s + k * k` を実行しようとしたところで, 代入されたことがなかった `s` を参照しようとした

教訓 (for 文のあるあるなエラー)

- for 文の中である変数 (今回の `s`) を, 自分自身で更新する (`s = ... s ...`) ことはよくある
- そういう時, for 文に先立ってその変数に代入しておかないといけないのでは? と反射的に思えるように
  - (多分)NG:

```
for ...
    s = ... s ...
```

  - 必要:

```
s = ...
for ...
    s = ... s ...
```

数式から for 文への翻訳作業例 (1)

- 方法 1: 手計算する自分の姿をイメージ:

$$1^2 + 2^2 + \cdots + n^2$$

$$\begin{aligned} s &= 0 \\ s &= s + 1^2 (= 1) \\ s &= s + 2^2 (= 5) \\ s &= s + 3^2 (= 14) \\ &\cdots \\ s &= s + n^2 \end{aligned}$$

($\rightarrow$ k を 1 から n まで動かして $s = s + k^2$ を繰り返すと見抜く)

```
1 s = 0
2 for k in range(1, n+1):
3     s = s + k * k
```

数式から for 文への翻訳作業例 (2)

- ややこしい場合は，漸化式を元にするのはいい方法．一般に，

```
s = c
for i in range(0, n):
    s = f(s)
```

の終了後の s は，

$$\begin{cases} s_0 &= c, \\ s_{i+1} &= f(s_i) \end{cases}$$

という数列の，s_n を保持している．

- 従って逆に，求めたいものを上記の漸化式に書きなおせば，for 文への翻訳は機械的:

数式から for 文への翻訳作業例 (2')

- 例:

$$s_n = 1^2 + 2^2 + \cdots + n^2$$

$\rightarrow$ 漸化式に翻訳

$$\begin{aligned} s_0 &= 0 \\ s_{i+1} &= s_i + (i+1)^2 \end{aligned}$$

$\rightarrow$ プログラムに翻訳

```
1 s = 0
2 for i in range(0, n):
3     s = s + (i + 1) ** 2
```

条件分岐: if 文

- 文法:

```
if ⟨ 式 ⟩:
    ⟨ 文 ⟩
     ...
    ⟨ 文 ⟩
else:
    ⟨ 文 ⟩
     ...
    ⟨ 文 ⟩
```

- 実行規則:
  1. ⟨ 式 ⟩ を評価
  2. 結果が「真」であれば，`else:` 以前の ⟨ 文 ⟩ ··· ⟨ 文 ⟩ を，そうでなければ `else:` 以降の ⟨ 文 ⟩ ··· ⟨ 文 ⟩ を順に実行
- `else:` 部分は省略可 (何もしない)

if 文の例 (1)

- 小さい方を返す

```
def smaller(x, y):
    if x < y:
        return x
    else:
        return y
```

- 3 の倍数と 3 のつく数を表示する

```
def nabeatsu():
    for i in range(50):
        if i % 3 == 0 or 30 <= i < 40:
            print(i)
```

if 文の例 (2)

- 素数を判定 (n が素数 $\Longleftrightarrow$ $2, \cdots, (n-1)$ のどれでも割り切れない)

```
def prime(n):
    if n == 1:
        return 0
    for i in range(2, n):
        if n % i == 0:
            return 0  # 割り切れたから素数じゃない
    return 1
```

- $n-1$ までの素数を数える

```
def count_prime(n):
    c = 0
    for i in range(1, n):
        if prime(i):
            c = c + 1
    return c
```

- if の ⟨ 式 ⟩ 部では，0 以外の全ての数が「真」とみなされる

3 つ以上に分岐する if 文

- 文法:

```
1 if ⟨ 式 ⟩:
2     ...
3 elif ⟨ 式 ⟩:
4     ...
5 elif ⟨ 式 ⟩:
6     ...
7 ...
8 else:
9     ...
```

- `elif` (`else if` の省略形だと思えば良い) をいくつでも書ける
- ここでも `else:` 部分は省略可 (何もしない)

break 文, continue 文, pass 文

- 概念的に大したことはないが一応

```
1  ⟨ 文 ⟩ ::=
2   return ⟨ 式 ⟩      # return 文
3  | x = ⟨ 式 ⟩        # 代入文
4  | import m          # import 文
5  | ⟨ 関数定義 ⟩      # 実は関数定義も文
6  | ⟨ 式 ⟩            # 実は式も文の一種
7  | for 文
8  | if 文
9  | break
10 | continue
11 | pass
```

break, continue, pass

- break → for 文 (全体) をそこでやめる

```
1  for i in range(1, 8):
2      if i % 3 == 0:
3          break
4      print i
5  → 1
6  2
```

- continue → for 文の「現在の繰り返し」をそこでやめる

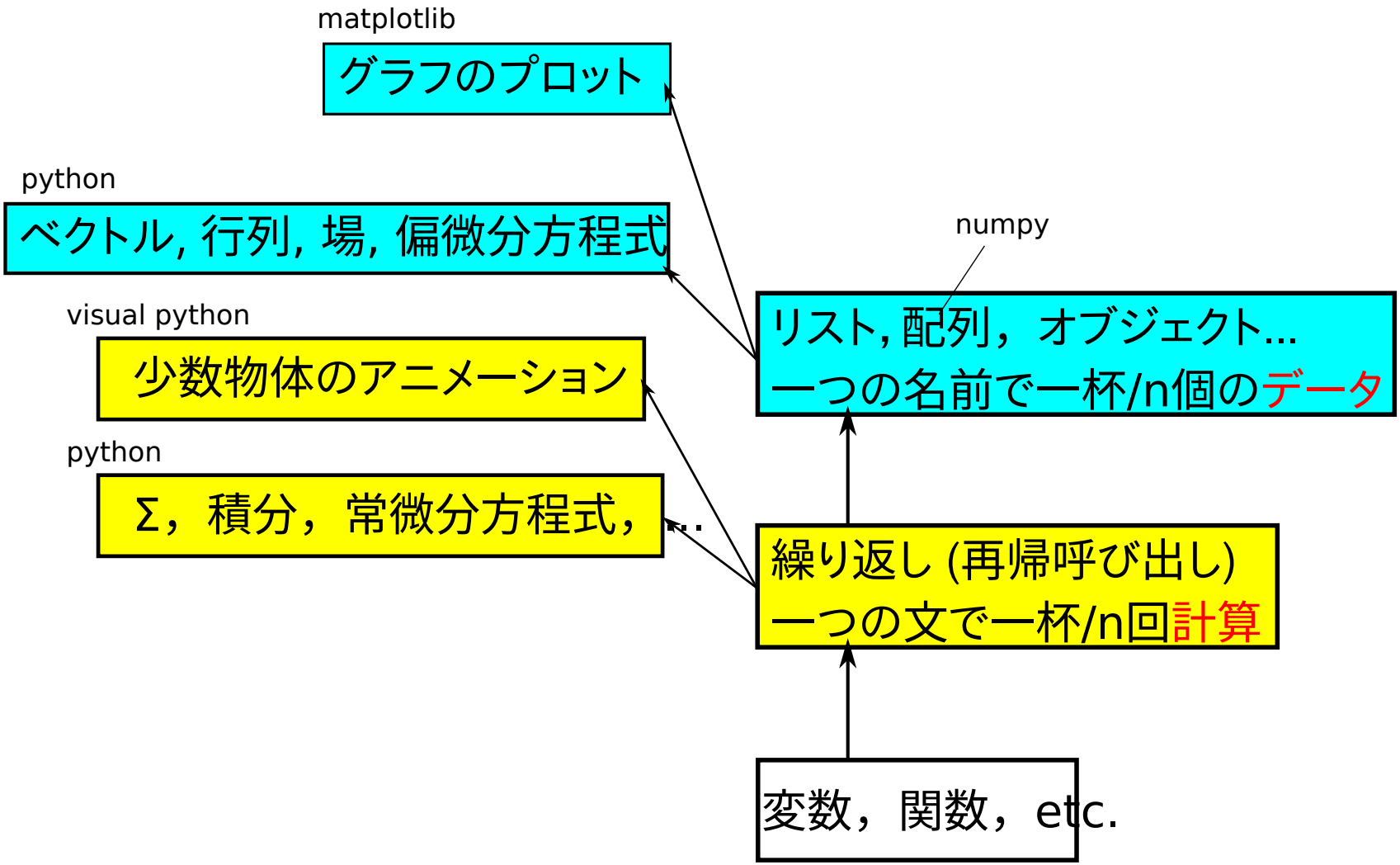
```
1  for i in range(1, 8):
2      if i % 3 == 0:
3          continue
4      print(i)
5  → 1
6  2
7  4
8  5
9  7
```

Contents

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
  - 2 次元ベクトルなら変数二つ，3 次元ベクトルなら変数三つ，. . . 100 次元なら. . . ?
- 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
- Python に備わる，複合 (compound) データ
  - リスト
  - タプル
  - 文字列
  - 辞書 (説明せず)
  - オブジェクト

以降の roadmap


matplotlib
グラフのプロット
python
ベクトル, 行列, 場, 偏微分方程式
numpy
visual python
少数物体のアニメーション
リスト, 配列，オブジェクト...
一つの名前で一杯/n個のデータ
python
Σ，積分，常微分方程式，...
繰り返し (再帰呼び出し)
一つの文で一杯/n回計算
変数，関数，etc.

Contents

リスト

- 文法 (式の一種):

```
1 ⟨ 式 ⟩ ::=
2   数
3 | x
4 | ⟨ 式 ⟩⟨ 演算子 ⟩⟨ 式 ⟩
5       ...
6 | [ 式, 式, ..., 式 ] # リストを作る
7 | 式 [式]              # 要素をアクセス
```

- [$E_0, E_1, \cdots, E_{n-1}$] : $E_0, \cdots, E_{n-1}$ (それぞれを計算した結果) を一本に束ねた値 (リスト)
- $L[i]$: L (を計算した結果のリスト) の i (を計算した結果) 番目の要素 (ただし最初の要素は「0 番目」)

リスト：例

```
1 import math
2 x = 20
3 a = [ 1, 1+2, math.cos(0.0), x ]
```

```
1 a
2 → [1, 3, 1.0, 20]
```

```
1 len(a)
2 → 4
```

```
1 a[1]
2 → 3
```

```
1 a + [1.2, 3.4]
2 → [1, 3, 1.0, 20, 1.2, 3.4]
```

```
1 [ a, a, a ]
2 → [[1, 3, 1.0, 20], [1, 3, 1.0, 20], [1, 3, 1.0, 20]]
```

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

```
1 def f(l):
2   return l[0]
3
4 f([1,2,3])
5 → 1
```

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

```
1 a = [ 2, 3, 5 ]
2 a.append(7) # 末尾に追加
3 a
4 → [2, 3, 5, 7]
```

ので, 繰り返し (for) を使ってとても長いリストを作ることもできる

```
1 a = []
2 for i in range(0, 10000):
3   a.append(i)
4 a
5 → [0, 1, 2, ..., 9999]
```

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

(書き換え)

```
a[2] = 50
a
→ [2, 3, 50, 7]
```

(削除)

```
del a[1]     # a[1]を削除．以降の要素は前にずれる
a
→ [2, 50, 7]
```

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

```
1 b = [ a, a, a ]
2 b
3 [[2, 50, 7], [2, 50, 7], [2, 50, 7]]
```

```
1 a[2] = 30    # 注目
2 b
3 [[2, 30, 7], [2, 30, 7], [2, 30, 7]]
```

```
1 dir(a)       # リストにできることは?
2 → [ ...  'append', 'count', 'extend', 'index',
          'insert', 'pop', 'remove', 'reverse', 'sort']
3 help(a.reverse)
4  ...
```

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:

```
1 ⟨ 式 ⟩ ::=
2       ...
3 | [ 式 for 名前 in range(式, 式) ]
```

- for 文同様，本当はもっと一般的 (後述).
- 意味: `[` E `for 名前 in range(式, 式) ]` は，あたかも

```
1 for 名前 in range(式, 式):
2   E
```

実行し，E を計算した結果を全てリストにする

- 例を見たほうが早い

リスト内包表記：例

```
1 [ x * x for x in range(0, 5) ]
2 → [0, 1, 4, 9, 16]
```

```
1 sum([ x * x for x in range(0, 5) ])
2 → 30
```

(sum はリストの和を取る関数)

それぞれ以下のようにしても同じことだがずっと簡潔

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

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

(注: s += E は s = s + E と同じ意味)

Contents

タプル

- 文法:

```
1 ⟨ 式 ⟩ ::=
2       ...
3 | 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

```
1 a = (1.1, 2.2, 3.3)
2 a.append(4.4) # NG
3 del a[1]      # NG
4 a[1] = 22     # NG
5 a = (4.4, 5.5) # OK
6 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 ≈ リストを不便にしただけ?

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く

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

- 注: 「二つの値を返す」という言い方はあまり正しくない. 二つの値を組にした一つの値 (= タプル) が作れる
- 変数に結果を受け取るときもこんなふうに書ける

```
1 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
  - これも OK だったりするので...

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

Contents

文字列

- 文法:

```
1 ⟨式⟩ ::=
2       ...
3 |     "文字文字 ...文字"
4 |     ’文字文字 ...文字’
5 |  """文字文字 ...文字"""
6 |  ’’’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 = ’"Hi!", said he.’`
    - `b = "Obama’s speech"`
  - 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,

```
1 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,

```
1 print("x = %s" % x)
```

のようにしてわかりやすく表示する

- 例

```
1 import math
2 y = math.cos(3.14)
3 print("cos(3.14) = %s" % y)
4 cos(3.14) = -0.999998731728
```

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

```
1 import math
2 x = 2.3
3 print("exp(%s) = %s" % (x, math.exp(x)))
4 exp(2.3) = 9.97418245481
```

文字列への値の埋め込み：規則

- 一般に，

```
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, あまり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, 多相性) 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

Contents

for 文の正体に近づく

- これまで for 文は以下の形であると言ってきた

```
for 名前 in range(式, 式):
    文
    ...
```

- 実はより一般的:

```
for 名前 in 式:
    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
- どの場合も, 「(リスト, タプル, 文字列などの) 各要素に対して文 ... を実行」という意味になる
- 実は, range(a, b) は [a, $a+1$, ...$b-1$] というリストを作る関数に過ぎなかった

for 文の正体に近づく

- 例:

```
1  for x in [ 2, 3, 5 ]:
2      print(x)
3  for x in "hello":
4      print(x)
5  2
6  3
7  5
8  h
9  e
10 l
11 l
12 o
```

タプルのリスト

- 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

```
1 A = [ (0,1), (2,3), (4, 5) ]
2 for x,y in A:
3     print(x + y)
4 1
5 5
6 9
```

- 以下のような文が許されることの拡張と思えば自然

```
1 x,y = タプル
```

zip : リスト, リスト → タプルのリスト

- 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

```
1 X = [ 1, 2, 3 ]
2 Y = [ 1, 4, 9 ]
3 zip(X, Y)
4 → [ (1,1), (2,4), (3,9) ]
```

```
1 for x,y in zip(X, Y):
2     print(x + y)
3 2
4 6
5 12
```

enumerate：場所を見ながらリストを処理

- 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

```
1 def find_space(s):
2   for i,c in enumerate(s):
3     if c == ' ':
4       return i
5   return -1
6
7 find_space("hello world")
8 → 5
```

Contents

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.
- 新しい種類 (class) のオブジェクトを自分で定義することもできるが，ここでは説明しない

オブジェクトについて知っておくべき最低限の事 (1)

- みんなみんな実はオブジェクト
  - Visual Python の vector, sphere, arrow, ...
  - (これから出る) numpy の「配列」
  - 実はリストやタプルもオブジェクト
- オブジェクトに対してできること
  1. メソッド (≈ 関数を呼び出す)

```
a.append(x)
```

  2. 属性 (フィールド) に値をセット

```
c.pos = vector(1,2,3)
```

  3. 属性を参照 c.x

```
print(c.x)
```

クラスとオブジェクト: 本当に最小の例

- 仕組みについて詳細に知る必要はない
- vector, sphere, なども「どこかで」こんな風に定義されている，とだけ心に留めとけばよし

```
1  class nothing:
2      pass  # 何もしない
3  # nothing クラスのオブジェクトを作る
4  m = nothing()
5  # 属性値へ代入 (≈ 変数への代入)
6  m.x = 10
7  m.y = 20
8  # 属性値を参照
9  print(m.x + m.y)
10 # もちろんオブジェクトを入力に取る関数も書ける
11 def take_nothing(n):
12     n.x = n.x + 100
13
14 take_nothing(m)
15 print(m.x)
```

オブジェクトについて知っておくべき最低限の事 (2)

- オブジェクトには「種類 (vector, リスト, タプル, ...)」がある = あるクラスに属している
- 同じ名前のメソッドでも, クラスが異なれば動作 (定義) が異なる
- $a + b$ とか，普段何気なくやっている動作も，クラスが異なれば動作が異なる
- これは
  1. 便利で強力 (数学でも+は数, ベクトル, 行列, etc. で定義が異なるが名前は同じ)
  2. 一方，ちゃんと意識しないと混乱や理解を曖昧にするもと
- 多くの場合その名前にふさわしい「自然な」動作が定義されていると期待する一方で，それらが「ひとりでに」「書き手の意図を読んで」行われるわけでは決して無いことを理解しよう

同じメソッドの名前でも, class が異なれば動作が異なる

- 例: append

```
1  import random
2  from vpython import curve,vector
3  def rnd():
4      return random.random()
5  l = []
6  c = curve()
7  for i in range(10):
8      l.append(vector(rnd(), rnd(), rnd()))
9  for i in range(10):
10     c.append(vector(rnd(), rnd(), rnd()))
```

+できえも

```
1 from visual import *  # vector
2 import numpy as np    # array
3 print(1 + 2)  # 足し算
4 print([1,2,3] + [4,5,6]) # リストの連結
5 print(vector(1,2,3) + vector(4,5,6)) # ベクトルの足し算
6 print(np.array([1,2,3]) + np.array([4,5,6])) # 同上
```

+できえも

```
from visual import *  # vector
import numpy as np    # array
print(1 + 2)  # 足し算
print([1,2,3] + [4,5,6]) # リストの連結
print(vector(1,2,3) + vector(4,5,6)) # ベクトルの足し算
print(np.array([1,2,3]) + np.array([4,5,6])) # 同上
```

```
3
[1,2,3,4,5,6]
<5, 7, 9>
[5 7 9]
```

*も

```
1 from visual import *  # vector
2 import numpy as np    # array
3 print(3 * 4) # 掛け算
4 print([1,2,3] * 5) # リストの繰り返し
5 print(vector(1,2,3) * 10) # ベクトルのスカラ倍
6 print(100 * vector(1,2,3)) # 同上
7 print(np.array([1,2,3]) * 1000) # 同上
8 print(10000 * np.array([1,2,3])) # 同上
```

*も

```
from visual import *  # vector
import numpy as np    # array
print(3 * 4) # 掛け算
print([1,2,3] * 5) # リストの繰り返し
print(vector(1,2,3) * 10) # ベクトルのスカラ倍
print(100 * vector(1,2,3)) # 同上
print(np.array([1,2,3]) * 1000) # 同上
print(10000 * np.array([1,2,3])) # 同上
```

```
12
[1,2,3,1,2,3,1,2,3,1,2,3,1,2,3]
<10, 20, 30>
<100, 200, 300>
[1000 2000 3000]
[10000 20000 30000]
```

もちろん**も

```
1 from visual import *  # vector
2 import numpy as np    # array
3 print(3 ** 4) # 3の4乗
4 print([1,2,3] ** 20) # 未定義
5 print(vector(1,2,3) ** 100) # 未定義
6 print(np.array([1,2,3]) ** 4) # 各要素の4乗
```

もちろん**も

```
from visual import *  # vector
import numpy as np    # array
print(3 ** 4) # 3の 4乗
print([1,2,3] ** 20) # 未定義
print(vector(1,2,3) ** 100) # 未定義
print(np.array([1,2,3]) ** 4) # 各要素の 4乗
```

```
81
# エラー
# エラー
[ 1 16 81]
```