アルゴリズム入門

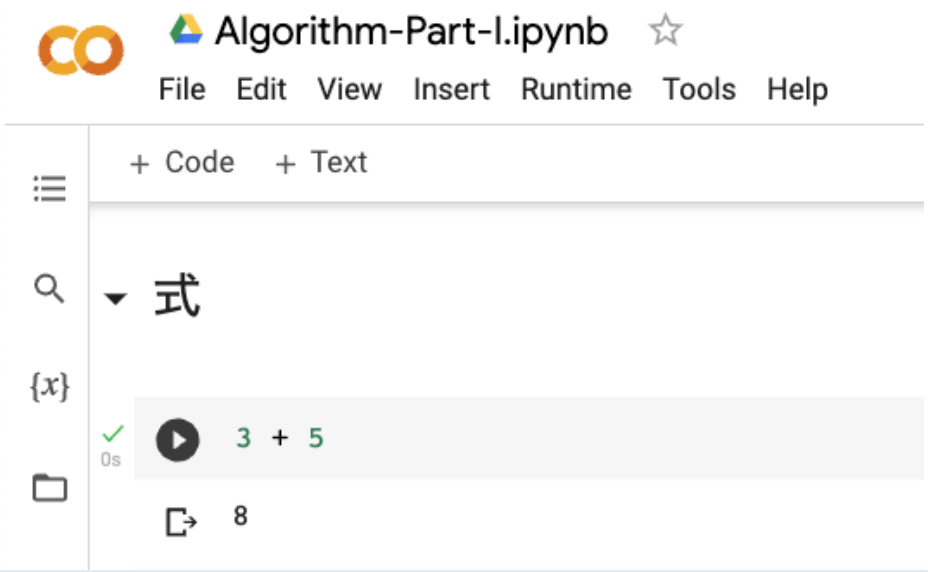
金子知適

東京大学教養学部教授 (kaneko@acm.org)

授業内容の紹介

目次

Part I, II

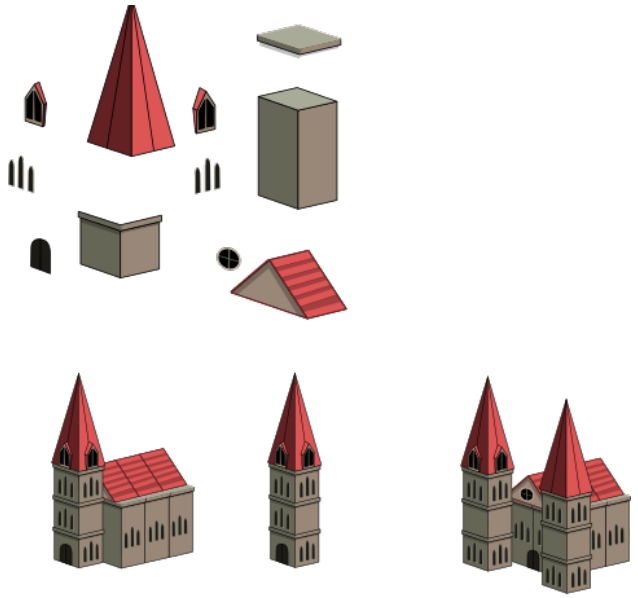


概念

- セル
- 式
- 値 (あたい) — 式の評価結果

Part I, II の目標

- 能力: Colab で，5行くらいのプログラムを，おおむね 思い通りに書ける
  - 実行結果を予想
  - 典型エラーメッセージの解読
- 体験: 単純な規則やパターンの組合せで，複雑な計算を構築・管理



計算手順の記法: 関数的 v.s. 命令的 (c.f. 「情報」 5.3 – 5.4章)

Part I: **理解しやすいこちらから**

関数的: 式と評価

- 式を評価することで計算が進行
- 式の意味は，場所を変えても同じ
  ➔ 分かりやすい

例: 算術式

- (3+5)*4 + 2 + sin(3+5)
- $f(x) = x^2 + 3x - 5$

目標

計算したい内容を，上手に**関数**や**変数**で整理して表現する．計算は，数値だけでなく文字などにも拡大．

注: 現実のプログラム言語は，様々な型の複合．言語の使い方 (style) も重要．
(c.f. 情報教科書 p. 131 コラム)

Part II **(今はいったん忘れる)** ▸ pp. 18–

命令的: 文の逐次実行

- 命令を順に実行
- ハードウェアの処理手順に近い
- 命令の結果は文脈に依存
  ➔ 文脈の共有 (=相互理解) が重要

情報の八十八夜問題, ED21 はこちら．

文脈の例

- 前に3歩進んで，右を向いて，さらに2歩すすめ．そこで地面を掘れ
- 今日から消費税は3%だよ．
  – 税抜き100円の品物は103円
- 今日から消費税は10%だよ．
  – 税抜き100円の品物は110円

式と評価

式 (1) 算術式　評価結果が数 (整数 int または浮動小数float) になる式

評価 (計算) すると値を得る．同じ式ならどの場所でも同じ意味.

算術演算の例

例1

$$\underbrace{\underbrace{(3+5)}_{8} * 2}_{16}$$

例2

$$\underbrace{-\underbrace{(-5)}_{-5}}_{5}$$

Q. ++5を評価した値は?

式とは

- **原始式** —今回は— 数 e.g., 3, 5, 3.14
  (評価すると自分自身になるもの)
- **式**と**演算子**の**適切な**組合せ e.g., (3+5), (1+(2+3))

(用語) 部分式: 式に含まれる式の呼称

数に対する演算子

2項演算子†		単項演算子‡	
+	加算	-	負号
-	減算	+	正号?♣
*	乗算		
/	除算		
//	整数除算		
%	剰余		
**	べき乗		

(♣ 正号は普通は使わない)

†2項演算子

（式 ⟨演算子⟩ 式）

‡単項演算子

（⟨演算子⟩ 式）

自明な場合はカッコを省略できる (詳細省略)

式と評価

式 (2) 文字式

評価結果が文字列になる式

評価 (計算) すると値を得る．同じ式ならどの場所でも同じ意味．

文字演算の例

例1

$$\underbrace{\underbrace{(\texttt{'a'+ 'b'})}_{\texttt{'ab'}} + \texttt{'c'}}_{\texttt{'abc'}}$$

例2

$$\underbrace{\underbrace{(\texttt{'hail' + '2'})}_{\texttt{'hail2'}} + \texttt{'you'}}_{\texttt{'hail2you'}}$$

式とは

- **原始式** e.g., 数，**文字(列)** (New!)
  e.g., `'a'`, `"a"`, `'hello'`, `'X'`, `'99'`
  (評価すると自分自身になるもの)
- **式**と演算子の**適切な**組合せ e.g., `'hello'+'world'`

(どちらでも良い) ' single quote, " double quote

(似ているが違う) ~~‘ backquote~~

文字列に対する演算子

2項演算子 +
（式 + 式）自明な場合はカッコを省略できる (詳細省略)

Q. 何で + が文字列の結合になるの?
. . . . . A. Python設計者が (便利なように) 定めた

補足: 2項演算子の左右の空白は，読みやすさのため

型

型 (type)

式は，型 (かた, type) を持つ.
演算子の意味 (動作) は型毎に定義.
(雑) 数と文字列を区別する

type 演算子

Python に型を質問してみよう

- `type(3)` ➔ `int`
- `type(3+5)` ➔ `int`
- `type(3.1415)` ➔ `float`
- `type('abc')` ➔ `str`

Q. 同じ + でも，3+5と'3'+'5'の結果が変わる理由は?

型が違うから．+ 演算子は，型に応じた計算を行う.

Q. 5-3 は2だけど，'5'-'3' は?

TypeError
あらかじめ定義された型についてのみ演算可能.

おまけ: `'hello!'*3` は
Pythonでは 定義されている.
(多くの言語ではエラー)

関数

関数:

入出力の対応関係を表現

数学の例

$$f(x) = x^2 + 100$$

のとき

- $f(0)$ ➔ 100
- $f(5)$ ➔ 125

Python の例

```
def f(x):
    return x**2 + 100
```

というセルを**実行後**

- `f(0)` ➔ 100
- `f(5)` ➔ 125

関数定義の基本パターン

```
def 関数名(x):
    return xを使う式
```

厳密さ注意

- パターン中の `def : return` は一字一句このまま
- `return` の左に，**スペース** (空白文字) を**2つ**

拡張: 引数 (ひきすう，～ 変数) 名は `x` でなくても良い，複数でも良い

関数定義の基本パターン (改)

```
def 関数名(仮引数1, 仮引数2, ...):
    return 仮引数を使う式
```

```
def add(a, b):
    return a + b
```

式と評価と関数

式と評価: ここまでのまとめ

- **原始式** e.g., 数，文字列 (評価すると自分自身になるもの)
- **式**と演算子の**適切な**組合せ
- **変数**, **関数** (new!)

式を評価して**値** (あたい) を得る.

四則演算

$$\underbrace{\underbrace{(3+5)}_{8} * 2}_{16}$$

関数+四則演算

```
def add(a, b):
  return a+b
```

$$\underbrace{\underbrace{\mathtt{add}(3,5)}_{a+b|_{a=3,b=5}} * 2}_{16}$$

変数 (引数) + 四則演算

$$a \leftarrow 3$$

$$\underbrace{\underbrace{(\underbrace{a}_{3}+5)}_{8} * 2}_{16}$$

変数を使う関数

関数のパターン改[2]

```
def 名前(仮引数1, 仮引数2...):
  変数1 = 式
  変数2 = 式
  ...
  return 式
```

- return の行の前に変数1, 2..を定義
- return する式には，仮引数に加えて変数1, 2..を利用可能
- **厳密さ注意** def につづく行は行頭にスペースを 2†

† Pythonでは 4 が推奨だが，この授業では 2 で統一

注: しばらくの間「変数」を値に名前をつけたもの (定数) として扱う.

例題: 三角形の面積 (ヘロンの公式)

3辺の長さ a, b, c に対応する三角形の面積を計算する関数 triangle_area(a, b, c) を作成せよ.
面積は $s = (a+b+c)/2$ として $\sqrt{s(s-a)(s-b)(s-c)}$ である.

回答例

```
def triangle_area(a, b, c):
  s = (a + b + c) / 2.0
  return ... ..... # ここにs,a,b,cを使った面積の式を書く
```

式と評価　論理式

式 (3) 論理式　評価結果が真偽値になる式

評価 (計算) すると値を得る．同じ式ならどの場所でも同じ意味．

論理演算の例

例1

$$\underbrace{\underbrace{(\texttt{True and False})}_{\texttt{False}}\ \texttt{or True}}_{\texttt{True}}$$

例2

$$\underbrace{\underbrace{(5<3)}_{\texttt{False}}\ \texttt{or}\ \underbrace{(3<5)}_{\texttt{True}}}_{\texttt{True}}$$

真偽値の型 (type) は **bool**

式とは

- 原始式: 数，文字(列)，**真偽値** `True`, `False`
  (評価すると自分自身になるもの)
- **式**と演算子の**適切な**組合せ `and`, `or`, `not`, `<`, `==`

論理演算　bool → bool

(論理式 `and` 論理式), (論理式 `or` 論理式),
(`not` 論理式)

自明な場合はカッコを省略できる (詳細省略)

比較演算　*any* → bool

(式 `<` 式), (式 `>` 式), (式 `<=` 式), (式 `>=` 式)
(式 `==` 式), (式 `!=` 式),

等号比較は，2文字重ねることに注意．
1文字の `=` は別用途に利用

比較演算子と 将来の 注意点 (やや高度)

他言語は試験範囲外

Python

Python の比較演算子は連鎖可能
(多項演算子, 親切)

$$\underbrace{5 > 3 > 1}_{\texttt{True}}$$

type 演算子

- `type(True)` ➔ `bool`
- `type(True+1)` ➔ `int` .....おや?†
  Q. なぜ**エラーにならない**?

型変換

- `int('777')` ➔ `777` .....便利!
- `str(777)` ➔ `'777'`
- `int(True)` ➔ `1` .....時に**罠**
- `int(False)` ➔ `0`

他の言語での注意

ほとんどの言語で,
比較演算子は**2項演算子**

失敗例

$$\underbrace{\underbrace{5 > 3}_{\texttt{True}} \ (\to 1)^{\dagger} \ > 1}_{\texttt{False?!}}$$

正しい記述

2項演算子に分解して構成 (冗長)

$$\underbrace{\underbrace{5 > 3}_{\texttt{True}} \ \texttt{and} \ \underbrace{3 > 1}_{\texttt{True}}}_{\texttt{True}}$$

† 型変換は，暗黙に行われることがある (詳細省略)

関数の動作の自動テスト

関数を作ったら，テストする． (c.f. 工業製品の出荷検査)
入出力の具体例 (**テストケース**) を，等号演算子==で比較する

関数

```
def add(a, b):
  return a + b
```

道具: assert

試験範囲外

assert 式

式: Trueになると期待する論理式

テストケース

- add(1, 2) ➔ 3
- add(100, -20) ➔ 80

人が仕様 (≈ 模範解答) を考える

```
assert add(1, 2) == 3
assert add(100, -20) == 80
```

実行結果の理解

- 無言 ➔ 正常状態
- AssertionError ➔ 異常，**要対処**

条件判断

関数作成のパターン改$^{2+\texttt{if}}$

```
def 名前(仮引数1, 仮引数2...):
  変数1 = 式
  変数2 = 式
  ...
  if ⟨条件 論理式⟩:
    return 条件が真の場合の式
  else:
    return 条件が偽の場合の式
```

厳密さ注意

- if や else の行末には : (コロン) (必須，定型)
- if の前は，空白文字 **2つ**
- return の前は，空白文字 **4つ**

重要: 空白文字の数の差分が2

復習 ◂ 論理式とは

例

絶対値　float → float あるいは int → int

```
def abs(x):
  if x >= 0:
    return x
  else:
    return -x
```

偶(even) 奇(odd)　int → str

```
def oddeven(x):
  if x % 2 == 0:
    return "even"
  else:
    return "odd"
```

条件判断 (2)

関数作成のパターン改^{2}+if-elif

```
def 名前(仮引数1, 仮引数2...):
  変数1 = 式
  ...
  if ⟨条件1 論理式⟩:
    return 条件1が真の場合の式
  elif ⟨条件2 論理式⟩:
    return 条件2が真の場合の式
  ...
  else:
    return 条件1,2,..,nが全て偽の場合の式
```

- (> 2) 分岐: else の前に，別の条件 elif で分岐することもできる
- 覚え方: elif $\approx$ else + if

関数作成のパターン改^{2}+if-nest

```
def 名前(仮引数1, 仮引数2...):
  変数1 = 式
  ...
  if ⟨条件1 論理式⟩:
    if ⟨条件1-1 論理式⟩:
      return 条件1と1-1が真のときの式
    else:
      return 条件1が真で1-1が偽のときの式
  else:
    return 条件1が偽の場合の式
```

木構造: return する代わりに，さらに if を重ねて (nest)，分岐できる

厳密さ注意

- if elif else 行末の : コロンを忘れない
- 外側の if と内側の if のインデント (行頭の空白文字数) の差が 2であること

式と評価

式 (4) 配列 (list)

評価 (計算) すると値を得る．どの場所でも同じ意味．

配列の演算の例

$$\underbrace{\underbrace{(\texttt{[3]} + \texttt{[1, 4]})}_{\texttt{[3, 1, 4]}} + \texttt{[1, 5]}}_{\texttt{[3, 1, 4, 1, 5]}}$$

```
a = [3, 1, 4, 1]
```

$$\underbrace{\texttt{a[0]}}_{3}$$

$$\underbrace{\texttt{len(a)}}_{4}$$

専門的には配列とlistは別のもの
(後日再訪 ‣ pp. 44–)

式とは

- **原始式** e.g., 数，文字(列)，真偽値，**配列**
  e.g., `[1,2,3]`, `[-99]`, `[1+2, 5]`, `["hey", "yo"]`, `[]`
  (式を角括弧でくくったもの．複数の式をコンマで区切ってくくることもできる)
- **式**と演算子の**適切な**組合せ e.g., +

配列に関する式

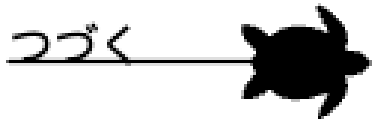
(式$_{\text{配列}}$ + 式$_{\text{配列}}$), 配列[式$_{\text{数}}$], len(式$_{\text{配列}}$)

文字列に関する式

(式$_{\text{文字列}}$ + 式$_{\text{文字列}}$), 文字列[式$_{\text{数}}$],
len(式$_{\text{文字列}}$)

文の逐次実行


```
turtle.pendown()
turtle.width(1)
turtle.forward(80)
```



Part IIへ

Part II

文の逐次実行

目次

計算手順の記法: 関数的 v.s. 命令的 (c.f. 「情報」 5.3 – 5.4章)

Part I これまで練習してきた ◂ 復習

関数的: 式と評価

- 式を評価することで計算が進行
- 式の意味は，場所を変えても同じ
  ➔ 分かりやすい

例: 算術式

- (3+5)*4 + 2 + sin(3+5)
- $f(x) = x^2 + 3x - 5$

目標

計算したい内容を，上手に**関数**や**変数**で整理して表現する．計算は，数値だけでなく文字などにも拡大．

重要: Part I のワークブックを取り組む際は，Part II の内容は使わないこと

Part II こちらを導入

命令的: 文の逐次実行

- 命令を順に実行
- ハードウェアの処理手順に近い
- 命令の結果は**文脈**に依存
  ➔ 文脈の共有 (=相互理解) が重要

- 八十八夜問題, ED21
- 前に3歩進んで，右を向いて，さらに2歩すすめ．そこで地面を掘れ
- 今日から消費税は3%だよ．
  — 税抜き100円の品物は103円
- 今日から消費税は10%だよ．
  — 税抜き100円の品物は110円

文の順序がとても重要

コンピュータの基本構成 (情報 8.1章)

教科書「情報」図8.1を思い出そう

CPU (中央処理装置)

万能チューリングマシン (テープ以外)
✓ 四則演算，短期記憶
レジスタ ∼ そろばん

メモリ (主記憶装置)

テープ相当:
✓ 任意の場所の読み書き

✓ チューリングマシンからの拡張部分 — 意図通りの操作をさせやすい

文

文

前から順に実行. 効果は「**状態**」(場所, 順序) **次第**.

print (new)

例1

```
print("もう終わったよ")
print("また課題が出たよ!")
```

例2

```
print("また課題が出たよ!")
print("もう終わったよ")
```

文法

print(式)

- `print(3)` 数値の表示
- `print(3+5)` 数値の表示
- `print("hello")` 文字列の表示

print(カンマで区切りの複数の式)

- `print("I have", 10**3, "yen")`
- `print("debug:  x = ", x)`

文

文

前から順に実行．効果は「**状態**」(場所, 順序) **次第**.

def (関数定義)

例1

```
# 前のセル
def add(a, b): # 関数定義
  return a+b
```

```
# 後のセル
add(2,5)
```

➔ 7

例2

```
# 前のセル
add(2, 5)
```

error !!

```
# 後のセル
def add(a, b): # 関数定義
  return a+b
```

定義より前にaddを使おうとすると失敗

補足:

- 上記の例は，上のセルから順に実行した状況が前提
- 実際には，Colab ではクリックした順に実行される
  - つまり，下のセルから実行することもできる
  - 各セルの左の [3] のような数字が実行順序を示す

文

文

前から順に実行．効果は「**状態**」(場所, 順序) **次第**.

= (変数への代入)

例1

```
1  x = 1
2  x = 100
```

実行後の x は 100

例2

```
1  x = 100
2  x = 1
```

実行後の x は 1

見た目注意

ほとんどのプログラミング言語で，= は等号ではなく代入記号.
心の中で = を ← と読み替えると良い.

c.f. 比較演算子は ==

使用注意

変数の書き換えはバグの元 ➔ しばらくは「基本パターン」のみで限定的に使う

変数 — 書き換え可能なメモ帳 —

変数とは

「値」を保持する．名前を持つ．値の**読み出し**と**書き込み**ができる．
計算手順の表記に利用 i.e., コンピュータの能力の仮定の一環

数学の変数と共通点

変数xについて..

- $x = 0$ のとき $\sin(x)$ は 0.0
- $x = \pi/2$ のとき $\sin(x)$ は 1.0

数学の変数との差異

逐次実行での値の変化(代入)

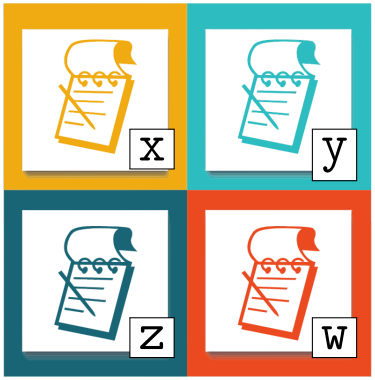
- x ← 1　　これ以降 xは1
- x ← x + 1　　これ以降 xは2
- x ← x + 1　　これ以降 xは3

歴史: ASCIIに ← がないので = で代用

文法

- 書き込み: 変数名 ← 式　　(**代入**)
- 読み出し: 式中の変数名は，値として評価される

Q. `x = x+1` って何? ..... A. $\underbrace{\texttt{x}}_{\text{変数名}} \leftarrow \underbrace{\texttt{x+1}}_{\text{式}}$



比喩: 変数 ∼ メモ帳

- 一番上の紙を読む
- 紙を一枚捨てて，次の用紙に値を書く

文と式

文

前から順に実行.
効果は「状態」(場所,順序) 依存.

- 式
- 変数 = 式　　(変数への代入)
- def ...　　(関数定義)
- return 式
- print(式) (new)
- if,then,else
- assert 式

式

評価(計算)すると値を得る. どの場所でも同じ意味.

原始式

- 算術
- 論理
- 文字列
- 配列

演算子と式の適切な組合せ, 関数

文と式はどちらも必要.
初めは, 決められたパターンで使い分ける (混ぜるな危険).

print文 と混ぜるな危険

評価 v.s. print

セル実行: 最後の式の評価† とそれまで print した内容が **混ざって** 表示される

†ただし None 以外

関数の return v.s. print

— return: 式の構成要素 —

```
def add(a, b):
  return a+b    # 基本パターン
```

- add(3, 5) ➔ 8
- add(3, 5) + 7 ➔ 15
- add(1000, add(3, 5) + 7)
  ➔ 1015

— print: 外部に出力 —

```
def add(a, b):
  print(a+b)    # 間違い
```

- add(3, 5) ➔ 8 (と表示)
- add(3, 5) + 7
  ➔ 8 (と表示)後にエラー
- add(1000, add(3, 5) + 7)
  ➔ 8 (と表示)後にエラー

print の結果は，外部 (ディスプレイ，プリンタ，テスター) に送られる．
プログラムから再度利用することはできない

for 文のくり返し実行

for の基本パターン (文)

```
for i in range(整数式):
  文1
```

- 意味　文1を，「整数式」回実行　◂ 式
- i はループカウンタ (他の文字も可)
- **厳密さ注意** 文1のインデントは厳密に 2†
  インデント ∼ 行頭の空白文字の数
- **厳密さ注意** 行末のコロン : を忘れずに
  (半角, ASCII文字)

† Python標準では 4 が推奨されているが，この授業ではスライドの紙幅の都合で 2 で統一

例　— 左右のコードは等価 —

```
print('Hi, there!')
print('Hi, there!')
print('Hi, there!')
```

```
for i in range(3):
  print('Hi, there!')
```

```
print(0)
print(1)
print(2)
```

```
for i in range(3):
  print(i)     # ループカウンタ i を式に利用
```

for 文のくり返し実行 — 文をブロックに拡張 —

for の基本パターン (ブロック)

```
for i in range(整数式):
  ブロック
```

- 意味　「**ブロック**」を，「整数式」回実行
- 定義「ブロック」
  インデントが揃った連続する文 (複数可)

例　筋力トレーニングメニュー — 各 for の制御範囲 —

```
for i in range(3):
  print('腕立て')
for i in range(3):
  print('腹筋')
```

腕立て
腕立て
腕立て
腹筋
腹筋
腹筋

```
for i in range(3):
  print('腕立て')
  print('腹筋')
```

腕立て
腹筋
腕立て
腹筋
腕立て
腹筋

for　2重ループ

定義拡張　ブロックの構成要素に，for 文とそのブロックを含める ➔ 多重ループ

for の基本パターン (2重ループ)

```
for i in range(整数式_外):      # 外のループカウンタ i
  ブロック外1
  for j in range(整数式_内):      # 内のループカウンタ j
    ブロック内
  ブロック外2
```

外側1回につき，内側のブロックを「式 (内)」回実行する．
➔ 内側のブロックは合計で，「式 (外) と式 (内) の積の回数」実行される．

```
for i in range(3):
  print('トレーニング始め!') ..........................  # ブロック外1
  for j in range(2):
    print(' 腕立て') ..................................  # ブロック内
    print(' 腹筋')                                        # 合計6回
  print('(5分休憩)') ..................................  # ブロック外2
```

for　2重ループとループカウンタ

定義拡張: ブロックの構成要素に，for 文とそのブロックを含める ➔ 多重ループ

for の基本パターン (2重ループ)

```
for i in range(整数式_外): #  外のループカウンタ i
  ブロック外1 ..... i の式を利用可
  for j in range(整数式_内): #  内のループカウンタ j
    ブロック内 ..... i, j の式を利用可
  ブロック外2 ..... i の式を利用可
```

例題: 次の結果を予想せよ

```
for i in range(3):
  for j in range(2):
    print(i+j)      # ブロック内
```

[重要!] for を使う関数 — 範囲の全てに何かする —

関数の基本パターン (for)

```
def 関数名(引数1,...):
  変数 = 式     # 初期値
  for i in range(..):
    変数 = 式     # 値を更新
  return 変数
```

例　0 から n-1 までの和

```
def sum_to_n(n):
  total = 0
  for i in range(n):
    total = total + i
  return total
```

考え方 — $n = 4$ の具体例, i.e., 0+1+2+3 —

部分和 s_i (0からiまでの和) で表現

```
def sum3():
  se = 0 # 空集合の和
  s0 = se + 0 # 0 (の和)
  s1 = s0 + 1 # [0,1] の和
  s2 = s1 + 2 # [0,1,2] の和
  s3 = s2 + 3 # [0,1,2,3] の和
  return s3
```

部分和 s_i をまとめて total (略記 t) で表現

```
def sum3():
  t = 0     # t は total の略記, 初期値 se
  t = t + 0 # t は s0 相当に
  t = t + 1 # t は s1 相当に
  t = t + 2
  t = t + 3
  return t     # t は s3 相当
```

Q. ところで, なぜ 0 を足すの?

A. 0 を無視できないケース, たとえば, $f(0)$ から $f(n-1)$ の和などに, このパターンを応用するため

[重要] ループ不変条件

(授業**範囲内**，試験範囲**外**)

関数の基本パターン (for)

```
def 関数名(引数1,...):
  変数 = 式        # 初期値
  for i in range(..):
    # (a) 更新 前 の i と変数の関係
    変数 = 式        # 値を更新
    # (b) 更新 後 の i と変数の関係
  return 変数
```

例　0 から n-1 までの和

```
def sum_to_n(n):
  total = 0
  for i in range(n):
    # (a) total = 0..i-1 の和
    total = total + i
    # (b) total = 0..i の和
  return total
```

Best practice: **ループ不変条件** (a), (b) どちらかをコメントに書こう! (平常点)

考え方: 3 = (4-1) までの合計,　i.e., 0+1+2+3

補助変数 s_i (0からiまでの和) の表現

```
def sum3():
  se = 0 # 空集合の和
  s0 = se + 0 # 0 (の和)
  s1 = s0 + 1 # [0, 1] の和
  s2 = s1 + 2 # [0, 1, 2] の和
  s3 = s2 + 3 # [0, 1, 2, 3] の和
  return s3
```

補助変数s_i を一つの変数 total で表現

```
def sum3():
  total = 0    # totalは se 相当
  total = total + 0 # total は s0 相当に
  total = total + 1 # total は s1 相当に
  total = total + 2
  total = total + 3
  return total    # total は s3 相当
```

演習　様々なパターン

例　0 から n-1 までの2乗和

```
def squared_sum_to_n(n):
  total = 0
  for i in range(n):
    #  (a) 0 (i == 0)
    #  (a) 0..i-1 の2乗和 (i > 0)
    total = total + i*i
    #  (b) 0..i の2乗和
  return total
```

例　整数の配列 seq の要素の和

```
def sum_of(seq):
  total = 0
  for i in range(len(seq)):
    #  (a) 0 (i == 0)
    #  (a) seq[i-1] までの和 (i>0)
    total = total + seq[i]
    #  (b) seq[0]..seq[i] の和
  return total
```

例　文字列の配列 seq の連結

```
def concat_of(seq):
  ans = ''
  for i in range(len(seq)):
    #  (a) '' (i == 0)
    #  (a) seq[i-1] までの連結 (i>0)
    ans = ans + seq[i]
    #  (b) seq[0]..seq[i] の連結
  return ans
```

例　配列の作成

```
def iota(n):
  ans = []
  for i in range(n):
    #  (a) [] (i == 0)
    #  (a) [0,..,i-1] (i>0)
    ans = ans + [i]
    #  (b) [0,..,i]
  return ans
```

for と if の組合せ — 範囲の一部を使う —

関数の基本パターン (for + if)

```
def 関数名(引数1,...):
  変数 = 式        # 初期値
  for i in range(..):
    if 式:
      # (a) 更新 前 の i と変数の関係
      変数 = 式        # 値を更新
      # (b) 更新 後 の i と変数の関係
  return 変数
```

ifブロックに拡張

◂ 関数とif

```
if 式1:
  ブロック1    # 式1が成立時の処理
elif 式2:
  ブロック2    # 式2が不成立の時の処理
...
else:
  ブロックe    # どの式も不成立の時の処理
```

注　elifやelseと対応するブロックは省略可

例　0 から n-1 までの奇数の和

```
def odd_sum_to_n(n):
  total = 0
  for i in range(n):
    # (a) 0..max(0,i-1)までの奇数の和
    if i % 2 == 1:
      total = total + i
    # (b) total は 0..iまでの奇数の和
  return total
```

例　配列seq の要素の最小値

```
def min_of(seq):
  ans = seq[0]
  for i in range(len(seq)):
    # (a) seq[max(0,i-1)]までの最小値
    if seq[i] < ans:
      ans = seq[i]
    # (b) ans は seq[i]までの最小値
  return ans
```

while　文のくり返し実行 — 回数指定無し —

while の基本パターン

```
while 論理式:
  ブロック
```

- 意味　ブロックを，「論理式」◂ が真のあいだ実行
- **厳密さ注意**　ブロックのインデントは厳密に **2**
  インデント ~ 行頭の空白文字の数
- **厳密さ注意**　行末のコロン : を忘れずに
  (半角, ASCII文字)

例: for と同じ動作

```
for i in range(3):
  print('Hi, there!')
```

```
cnt = 0
while cnt < 3:
  print('Hi, there!')
  cnt = cnt + 1
```

```
for i in range(3):
  print(i)
```

```
cnt = 0
while cnt < 3:
  print(cnt)
  cnt = cnt + 1
```

while　文のくり返し実行 — 回数指定無し —

while の基本パターン

```
while 論理式:
  ブロック
```

- 意味　ブロックを，「論理式」◂ が真のあいだ実行
- **厳密さ注意**　ブロックのインデントは厳密に **2**
  インデント ~ 行頭の空白文字の数
- **厳密さ注意**　行末のコロン : を忘れずに
  (半角, ASCII文字)

例　while が for より適する動作

整数 n (> 1) に対する $\lceil \log_2 n \rceil$ は?
つまり，2で何回わると1以下になるか?

```
def log2up(n):
  cnt = 0
  while n > 1:
    n = n // 2
    cnt = cnt + 1
  return cnt
```

while　文のくり返し実行 — 回数指定無し —

while の基本パターン

```
while 論理式:
  ブロック
```

- 意味　ブロックを，「論理式」◂ が真のあいだ実行
- **厳密さ注意**　ブロックのインデントは厳密に **2**
  インデント ~ 行頭の空白文字の数
- **厳密さ注意**　行末のコロン : を忘れずに
  (半角, ASCII文字)

無限ループ注意

3回挨拶しようかな?
..... cnt は減り続ける

```
cnt = 0
while cnt < 3:
  print('Hi, there!')
  cnt = cnt - 1 # +1 のうっかり
```

止まらない (花粉症のシミュレーター?)
..... 止めて!

```
while True:
  print('はくしょん')
```

将来，意図的に無限ループさせることもあるが，今は避ける

Part III

多次元配列，再帰，計算量

目次

前半のまとめと後半の学習内容

Part I, II で学習 済

- 式と評価
  演算子，関数，変数
- 文と制御構造 (if, for, while)
  - 変数の書き換え
  - ループ不変条件

1. 正しい (e.g., 教科書の) コードの動きを予想できる ..... **Part II 当面の目標**
2. 誤ったコードを修正できる
3. 思い通りにかける .....Part I 目標
   「王道」の書き方を選ぶ，
   簡潔な表現に推敲

Part III で これから 学習

- 配列の変更操作　慣れてきたので解禁．bug注意
- 多次元配列，リストのリスト，辞書
  できることを自然に拡大
- 再帰　複雑な対象を管理下に
- 計算量の考え方　計算結果が同じでもどちら得?

教科書の topics ≈ 鑑賞

- ライフゲームとアニメーション
- フラクタル
- 整列 (マージソート)

おまけ

- グラフを描く `matplotlib`
- 数値計算 → 機械学習 `numpy` など

楽しい (できることは広がる) が，必須ではない

数の表現 — 導入 —

整数 int

- 数学の整数
- Python では，範囲に制限なし
  (他の多くの言語では，32 bit または 64 bit)

浮動小数 float

- 実数の **有限桁数での近似表現**
- 64 bit — IEEE 754 (後日, 教科書) —
  128 bit や 32, 16 bit 表現もよく使われる

2進表現

10進	2進
0	0
1	1
2	10
3	11
4	100
8	1000

```
bin(49) → '0b110001'
```

0b 1 1 000 1

0b: 2進表現
1: $32=2^5$
1: $16=2^4$
1: $1=2^0$

2進表現

10進	2進
0	0
0.5	0.1
0.25	0.01
0.125	0.001
1/16	0.0001
0.75	0.11

10進 0.1 = 1/10 は，
2進表現で **循環小数**
0.0001100110011001..

有限精度

0.1を10回足しても，
1.0にならない．

数学との違い

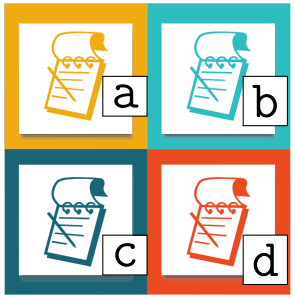
ほとんどの実数は(例 0.1) は表現したその場から誤差を伴う
精度は，演算過程でさらに失われうる (後日)

▸ pp. 90–

実体と参照

変数とは

「値」を保持する．名前を持つ．値の**読み出し**と**書き込み**ができる．
計算手順の表記に利用 i.e., コンピュータの能力の仮定の一環



比喩: 変数 ∼ メモ帳

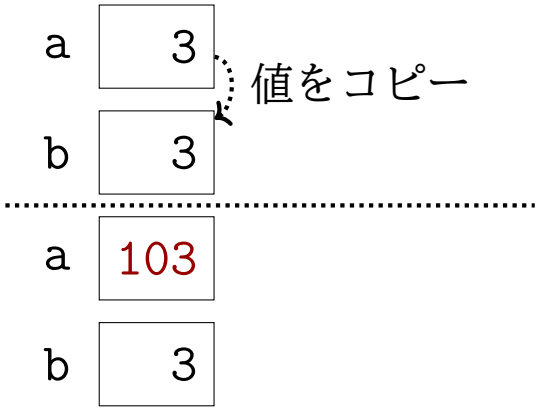
- 読む: 一番上の紙を読む
- 書く: 紙を一枚捨てて，次の用紙に値を記入

数

実体 (方便) を変数に記録
変数の複製 . . . . . 直観通り

```
a = 3
b = a
    3
```

```
a = a   + 100
    3
    103
```



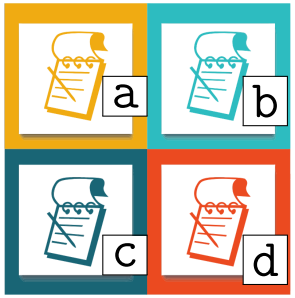
結果

- a → 103
- b → 3

実体と参照

変数とは

「値」を保持する．名前を持つ．値の**読み出し**と**書き込み**ができる．
計算手順の表記に利用 i.e., コンピュータの能力の仮定の一環



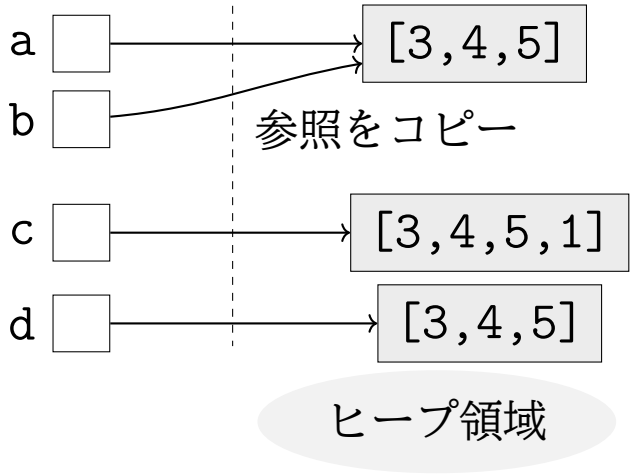
比喩: 変数 ∼ メモ帳

- 読む: 一番上の紙を読む
- 書く: 紙を一枚捨てて，次の用紙に値を記入

配列

「**参照**」 **(new!)** を変数に記録
変数の複製 . . . . . 参照の複製 **(!)**

```
a = [3,4,5]
b = a  # 要注意操作
c = a + [1]
d = [3,4,5]
```



比喩

- 実体 — 銀行口座のお金
- 参照 — 通帳

結果

- a ➔ [3,4,5]
- b ➔ [3,4,5]
- a==b==d ➔ True

変数 a，b は実体を共有

要区別: 「通帳」のコピー（b）と，「口座」の新規開設（c，d）

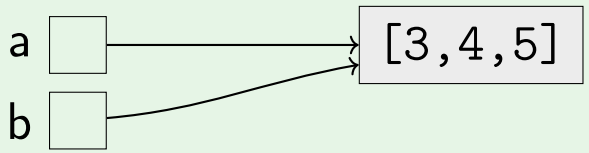
配列の破壊的変更

配列 seq の破壊的変更手段の例 — 実体に作用 —

- `seq[i] = 式` ..... 要素`seq[i]`を式の値に書き換え
- `seq.append(式)` ..... 末尾に式の値を持つ要素を追加
- `seq.sort()` ..... 要素を昇順に並び替え

変更前の状況

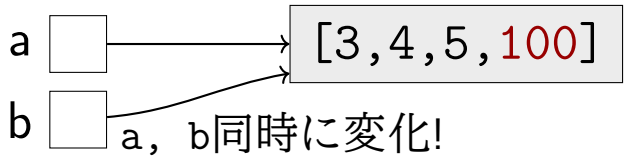
```
a = [3,4,5]
b = a
```



append

```
a.append(100)
```

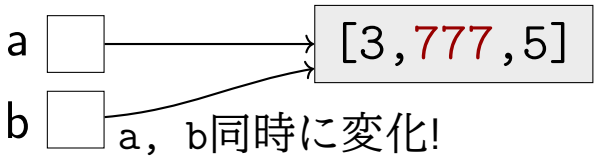
結果
- a ➔ [3,4,5,100]
- b ➔ [3,4,5,100]



要素書き換え

```
b[1] = 777
```

結果
- a ➔ [3,777,5]
- b ➔ [3,777,5]



要注意: 関数の引数も参照を複製 = 実体を共有

bug の要因なので，破壊的変更は **Part I,II では非推奨**．Part III 以降では解禁されるが慎重に．

配列とリスト

配列とリストの概念

(区別は試験範囲外)

多くのプログラミング言語では，配列 (array) とリスト (list) を別の「型」として取り扱う．入門のため授業では，両方に list を使う．

配列†

- 長さ: 基本固定
  appendで末尾に追加 (のみ) できる版も
- 高速

† 実用の Python では numpy.ndarray が標準的

リスト

- 長さ: 可変 e.g., [1,2,3] + [4]
  i.e., 要素の追加，リストの連結が可能
- 配列よりは遅い

配列としての利用 — 作成と操作 —

Step 1. 長さを指定して作成

```
a = [0, 0, 0]
```

想定: ジム会員の身長を，a で管理しよう

Step 2. 要素を書き換え

```
a[0] = 178
a[1] = 145
a[2] = 152
```

注:

- 上の例は，a = [178, 145, 152] と等価
- 長さ n (ソースコード記述時には未定) の配列を扱う準備 — 次ページで一般化

配列の作成 — 長さsizeの配列を作る 式

配列の作成方法

作成法1　教科書準拠

```
import ita # 1度だけ実行
```

```
ita.array.make1d(size)
```

試験ではこちら.
.....ただし，ita は，授業専用

作成法2　実用

```
[0 for _ in range(size)]
```

.....リスト内包表記 (試験範囲外)

作成法3　別解

```
[0] * size # 非推奨
```

.....2次元以降で罠があるので非推奨

応用 — 要素の初期値の指定

全部 1 の配列を作るには

作成法1　教科書準拠

```
ita.array.make1d(size, 1)
```

作成法2　実用

```
[1 for _ in range(size)]
```

.....リスト内包表記 (試験範囲外)

配列を作成する関数

配列を作成する関数の基本パターン

```
def 関数名(引数1,...):
  配列 = ita.array.make1d(長さ) # または ita.array.make1d(長さ, 初期値)
  for i in range(len(配列)):
    配列[i] = 式      #  各要素の値を設定
  return 配列
```

例1

長さ n で要素がすべて 1 の配列を返す関数 ones(n) を，基本パターンで 作成せよ

```
def ones(n):
  seq = ita.array.make1d(n)
  for i in range(n):
    seq[i] = 1
  return seq
```

例2

0 から n-1 までの整数を1づずつ要素として持つ配列を返す関数 iota_alt(n) を作成せよ

```
def iota_alt(n):
  seq = ita.array.make1d(n)
  for i in range(n):
    seq[i] = i
  return seq
```

.....ここまででいったん演習

配列の要素を書き換える関数 (手続き)

配列を書き換える関数 (手続き) の基本パターン

```
def 関数名(配列, 引数2,...):
  for i in range(len(配列)):
    配列[i] = 式      #  各要素の値を設定
# returnなし
```

手続き

returnしない関数を**手続き**と呼ぶことにする．手続きは，式としては使えない．式の外で計算に影響=**副作用**を及ぼす．便利だが慎重に使う．

計算のイメージ — 共通の黒板 (配列) に人々が (関数) 書き込む

例1　要素をすべて 1 に設定

```
def fill_one(seq):
  for i in range(len(seq)):
    seq[i] = 1
```

例2　要素を0 から順の整数に

```
def fill_iota(seq):
  for i in range(len(seq)):
    seq[i] = i
```

```
>>> a = [3,1,4]
>>> a
[3, 1, 4]
>>> fill_one(a)
>>> a
[1, 1, 1]  # 書き換わった
```

実行例の新表記法

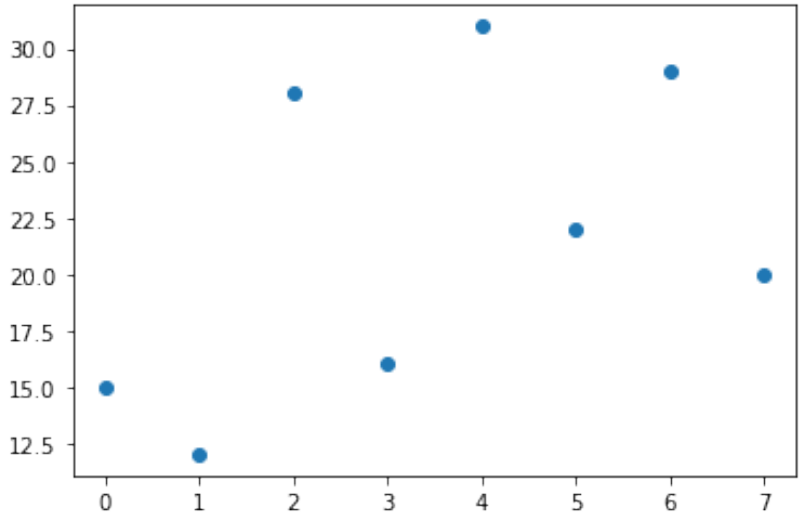
赤字の >>> は，Colab のセルの略記．
>>> a は，Colab のセルに a という式を入力し．続く灰色 [3,1,4] が Colab の応答 (評価結果) を示す．# 書き換わった のような付記は，教員による注釈．

配列の可視化 (試験範囲外)


```
import ita # どこかで1度実行
```

```
data = [15, 12, 28, 16, 31, 22, 29,
 ↪  20]
ita.plot.plotdata(data)
```

機能　(x,y) = (i, data[i]) に点を打つ



Best practice

- 可視化は便利なので，いつでもお勧め
- 注　itaは授業専用．一般には `matplotlib` を使う

2次元配列とリストのリスト

配列とリストの一般的な使い分け

2次元配列†

- 形状　長方形
- 大きさ　基本固定
- 要素の型　同一

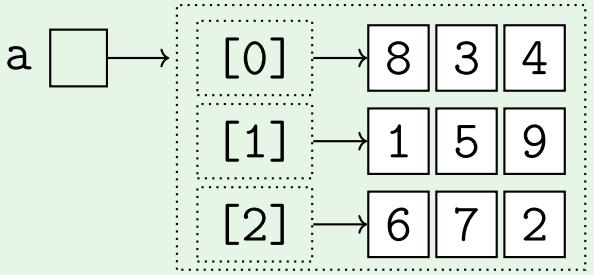
リストのリスト

- 形状　自由
- 大きさ　基本固定
- 要素の型　いろいろ

† 実用の Python では `numpy.ndarray` が標準的

2次元配列としての「リストのリスト」

```
a = [[8,3,4],
     [1,5,9],
     [6,7,2]]
```




```
>>> a[0]
[8,3,4]    # list
>>> a[0][0]
8    # int
```

使用例　行列，xy座標の高さ，ゲームボード

2次元配列の作成 — r行 c列の配列を作る 式

配列の作成方法

作成法1 教科書準拠

```
import ita # 1度だけ実行
```

```
ita.array.make2d(r, c)
```

試験ではこちら.
..... ただし，ita は，授業専用

作成法2 実用

```
[[0 for _ in range(c)] for _ in
↪  range(r)]
```

..... リスト内包表記 (試験範囲外)

罠 — 似て非なるもの —

```
[[0] * r] * c # ダメ
```

応用 — 要素の初期値の指定

全部 777 の配列を作るには

作成法1 教科書準拠

```
ita.array.make2d(r, c, 777)
```

作成法2 実用

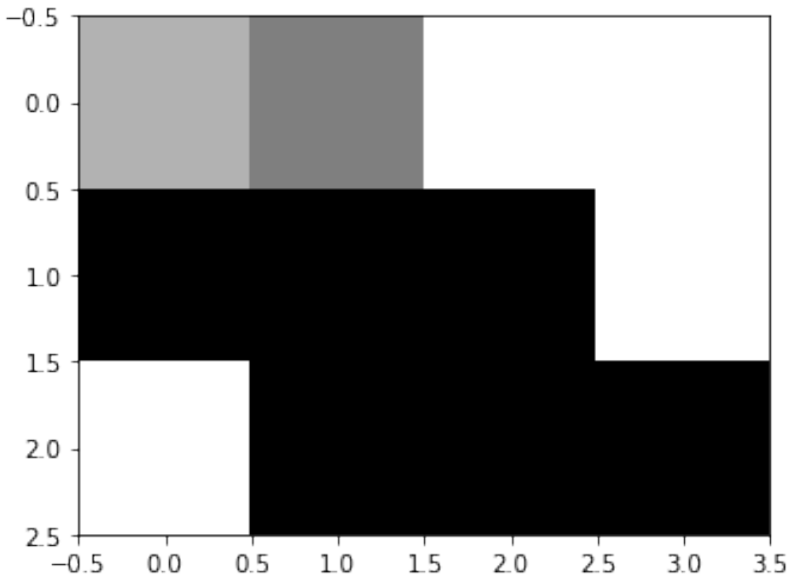
```
[[777 for _ in range(c)] for _ in
↪  range(r)]
```

..... リスト内包表記 (試験範囲外)

2次元配列の可視化


```
import ita # どこかで1度実行
```

```
data = [[0.7, 0.5, 1, 1],
        [0, 0, 0, 1],
        [1 ,0, 0, 0]]
ita.plot.image_show(data)
```



機能　(x,y) の明るさ ～ data[y][x]　　(0が黒，1が白)

Best practice

- 可視化は便利なので，いつでもお勧め
- 注: ita は授業専用．一般には matplotlib を使う

標準課題

この可視化の機能を利用して，ライフゲームを作ろう (教科書5章)

1 . . . . . 色をつけよう! ➔ 発展課題で

2次元配列　全要素を読む

復習

1次元配列の基本パターン

```
def 関数名(seq,引数2,...):
  変数 = 式 # 初期値
  for i in range(len(seq)):
    変数 = seq[i] を使う式
  return 変数
```

例　全要素の和

```
def sum_of(seq):
  total = 0
  for i in range(len(seq)):
    total = total + seq[i]
    # ループ不変条件 total は seq[i]までの和
  return total
```

2次元配列の基本パターン

```
def 関数名(seq2d,引数2,...):
  変数 = 式 # 初期値
  for r in range(len(seq2d)): # 行r
    for c in range(len(seq2d[0])): # 列c
      変数 = seq2d[r][c]を使う式
                 読む
  return 変数
```

2重ループになったが，本質は1次元と共通

例　要素の最小値

```
def min_in_mat(a):
  minimum = a[0][0]
  for r in range(len(a)):
    for c in range(len(a[0])):
      if minimum > a[r][c]:
        minimum = a[r][c]
      # 不変条件 a[r'][c'] 内の最小
      # r' < r or (r' = r, c' <= c)
  return minimum
```

別解　各行で最小値をとって，最後に総合する

2次元配列　全要素を書く

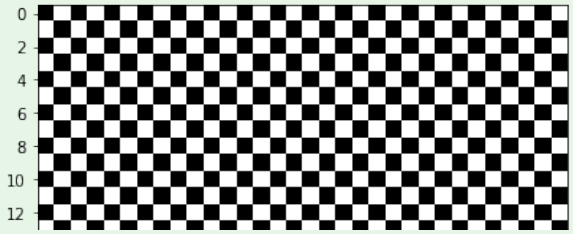
2次元配列の要素を書くパターン

```
def 関数名(seq2d, num_rows, num_cols):
  seq2d = ... # 作成(num_rows, num_cols)
  for r in range(len(seq2d)): # 行r
    for c in range(len(seq2d[0])): # 列c
      seq2d[r][c] = 式
        書きこみ先
  return seq2d
```

2重ループになったが，本質は1次元と共通

例　0,1を交互に書き込め

```
>>> rectangle_check(4,4)                                    1
[[0,1,0,1], [1,0,1,0], [0,1,0,1], [1,0,1,0]]                2
>>> ita.plot.image_show(                                    3
 ↪  rectangle_check(16,32) )
```



実装例1

```
def rectangle_check(r, c):
  img = ita.array.make2d(r, c, 0)
  for y in range(r):
    for x in range(c):
      if ....: # 1を書く条件
        img[y][x] = 1
      elif ....: # 0を書く条件
        img[y][x] = 0
  return img
```

実装例2

```
def rectangle_check(r, c):
  img = ita.array.make2d(r, c, 0)
  for y in range(r):
    for x in range(c):
      img[y][x] = (x + y) % 2
  return img
```

注: ワークブックの次の例題 gingham は，灰色が2種類で教科書より少し凝った柄

表の集計

演習　2次元配列の行と列を拡張し，合計を集計せよ

入力例

8	3	4
1	5	9
6	7	2

出力例

8	3	4	15
1	5	9	15
6	7	2	15
15	15	15	45

観察

copy2d	行の和
縦の和	総計

行の和 横方向

```
def rsum(a, r):
  total = 0
  for c in range(len(a[r])):
    total += a[r][c]
  return total
```

列の和 縦方向

```
def vsum(a, c):
  total = 0
  for r in range(len(a)):
    total += a[r][c]
  return total
```

実装例

```
def extended_table(a):
  b = ita.array.make2d(len(a)+1, len(a[0])+1)
  for r in range(len(b)):
    for c in range(len(b[r])):
      if r < len(a) and c < len(a[r]):
        b[r][c] = a[r][c] # copy
      elif r < len(a):
        b[r][c] = rsum(a, r) # yoko
      elif c < len(a[0]):
        b[r][c] = vsum(b, c) # tate
  b[-1][-1] = rsum(b, len(b)-1)
  return b
```

これから学ぶこと

- 複雑な内容をシンプルなプログラムで記述する
  ➔ 再帰や動的計画法 (教科書では11章，試験範囲外) は有用
- 複雑な内容を扱うので，失敗すると，予想外のことも起こる
- 正しい答えを出すプログラムでも，とても遅くなりうる
- ちょっとした工夫で，遅いプログラムが速くなることも
  ➔ 計算量の見積もりを学ぼう

再帰 recursion

再帰の位置づけ

- 複雑さを管理下に置く道具の1つ
  e.g., フラクタル，グラフの探索
- 反復 (`for`, `while`) と補完し合う

再帰と反復との関係

	反復得意	反復苦手
再帰得意	**(1)**	(2)
再帰苦手	(3)	-

再帰でも反復でもかける内容 (1) から始めて，再帰が得意とする内容 (2) に進む

再帰関数の基本パターン — 漸化式と対応させる —

例: $n \geq 0$ までの和

漸化式

$$\mathrm{sum}(n) = \begin{cases} 0 & n = 0 \text{ のとき} \\ n + \underbrace{\mathrm{sum}(n-1)}_{n\text{-}1\text{までの和}} & \text{それ以外} \end{cases}$$

```
def sumr(n):
  if n == 0: # basecase
    return 0
  else: # recursion step
    return n + sumr(n-1)
                  n-1までの和
```

(sumr の r は recursion から命名)

複数の表現例

再帰関数の基本パターン — 漸化式と対応させる —

例: $n \geq 0$ までの和 (再掲)

漸化式

$$\mathrm{sum}(n) = \begin{cases} 0 & n = 0 \text{ のとき} \\ n + \underbrace{\mathrm{sum}(n-1)}_{n\text{-}1\text{までの和}} & \text{それ以外} \end{cases}$$

```
def sumr(n):
  if n == 0: # basecase
    return 0
  else: # recursion step
    return n + sumr(n-1)
                 ^ n-1までの和
```

例: $n \geq 0$ までの和　別表現 (発展 — 末尾再帰)

漸化式

$\mathrm{sum}(n) = \mathrm{sum}'(0, n)$, where

$$\mathrm{sum}'(s, n) = \begin{cases} s & n = 0 \text{ のとき} \\ \mathrm{sum}'(s+n, n-1) & \text{それ以外} \end{cases}$$

変数 sは，「大きい方からの和」

```
def sum2(s, n):
  if n == 0: # basecase
    return s
  else: # recursion step
    return sum2(s+n, n-1)
                 ^ (s+n)+(n-1までの和)
```

再帰の注意

Maximum recursion depth exceeded

再帰関数は，basecase で止まるように実装する必要がある．
以下はダメな例で，無限再帰する

```
def inf():
  # basecase がない!
  return 1 + inf() # step
```

この関数を`inf()`と呼び出すと，

```
>>> inf()
Traceback (most recent call last):
  ...
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

というように，`RecursionError: maximum recursion depth exceeded`というエラーで処理を続行できなくなった旨表示される．

例題

例: 最大公約数

漸化式

$$\gcd(a,b) = \begin{cases} b & \text{if } a \text{ is a multiple of } b \\ \gcd(b, a\%b) & \text{otherwise} \end{cases}$$

実用 `math.gcd(a, b)`

```
def gcd(a, b):
  if a % b == 0: # basecase
    return b
  else:          # step
    return gcd(b, a % b)
```

例: 平方根の二分法 (「情報」)

漸化式

$$\mathrm{sqrt}(x,a,b) = \begin{cases} m & \text{if } |m^2 - x| < \epsilon \\ \mathrm{sqrt}(x, \underbrace{a, m}_{\text{左半分}}) & \text{if } m^2 > x \\ \mathrm{sqrt}(x, \underbrace{m, b}_{\text{右半分}}) & \text{otherwise} \end{cases}$$

実用 `math.sqrt(x)`

```
Epsilon = 0.0001 # 許容誤差
def sqrt(x, a, b):
  m = (a+b)/2.0
  if abs(x - m*m) < Epsilon:
    return m     # basecase
  elif m*m > x:  # step
    return sqrt(x, a, m)
  else:          # step
    return sqrt(x, m, b)
```

二分法

「情報5章」

$f(x) = 0$ の解を求める二分法

初期状態

区間 $[a_0, b_0]$
s.t. $f(a_0) < 0 < f(b_0)$

step $n \to n+1$, i.e., $[a_n, b_n] \to [a_{n+1}, b_{n+1}]$

中点$c_n = (a_n + b_n)/2$ を用いて，

$$(a_{n+1}, b_{n+1}) = \begin{cases} (a_n, c_n) & f(c_n) > 0 \\ (c_n, b_n) & \text{otherwise} \end{cases}$$

教科書「情報」図5.6を思い出そう

1回で範囲が半分に ➔ $\log_2$ (区間全体/目標精度) くらい

再帰的手続き

手続き (値を返さない関数，副作用で仕事) を再帰的に書くこともできる．漸化式とはもはや対応しないが，basecase と step という考え方は有用である．

nから1まで表示

```
def printnum(n):
  if n == 0:  # ~ basecase
    return
  else:   # ~ step
    # ここまでに n < x の x は表示済み
    print(n)
    printnum(n-1)
```

```
>>> printnum(3)
3
2
1
```

1からnまで表示

```
def printnum_inc(cur, n):
  if cur > n:  # ~ basecase
    return
  else:  # ~ step
    # ここまでに x < cur の x は表示済み
    print(cur)
    printnum_inc(cur+1, n)
```

```
>>> printnum_inc(1, 3)
1
2
3
```

反復と再帰 — 直線的な再帰は反復と対応

例: $n \geq 0$ までの和

漸化式

$$\mathrm{sum}(n) = \begin{cases} 0 & n = 0\text{のとき} \\ n + \underbrace{\mathrm{sum}(n-1)}_{\text{n-1までの和}} & \text{それ以外} \end{cases}$$

再帰と反復の関係 (再掲)

	反復得意	反復苦手
再帰得意	**(1)** 線形再帰	(2)
再帰苦手	(3)	-

再帰版 (再掲)

```
def sumr(n):
  if n == 0:
    return 0
  else:
    return n + sumr(n-1)
```

n-1までの和

等価な for の実装

```
def sumf(n):
  total = 0
  for i in range(n+1):
    # (a1) この時点で total は 0 (i == 0)
    # (a2) この時点で total は i-1 までの和
    total += i
    # (b) この時点で total は i までの和
  return total
```

反復と再帰 — 直線的な再帰は反復と対応

例: 平方根の二分法 (「情報」)

漸化式

$$\mathrm{sqrt}(x,a,b) = \begin{cases} m & \text{if } |m^2 - x| < \epsilon \\ \mathrm{sqrt}(x,a,m) & \text{if } m^2 > x \\ \mathrm{sqrt}(x,m,b) & \text{otherwise} \end{cases}$$

再帰と反復の関係 (再掲)

	反復得意	反復苦手
再帰得意	**(1)** 線形再帰	(2)
再帰苦手	(3)	-

再帰版 (再掲)

```
Epsilon = 0.0001 # 許容誤差
def sqrt(x, a, b):
  m = (a+b)/2.0
  if abs(x - m*m) < Epsilon:
    return m
  elif m*m > x:
    return sqrt(x, a, m)
                   左半分
  else:
    return sqrt(x, m, b)
                   右半分
```

反復版

```
def sqrt_iter(x, a, b):
  m = (a+b)/2.0
  while abs(x - m*m) >= Epsilon:
    if m*m > x:
      b = m # 次の区間[a,b] ← 今の左半分[a,m]
    else:
      a = m # 次の区間[a,b] ← 今の右半分[m,b]
    m = (a+b)/2.0
  return m
```

枝分かれを伴う再帰

いろいろな再帰

	関数	手続き
易 線形	sum(n) gcd(a,b), sqrt(x,a,b)	printnum(n)
枝分かれ 合流無し	木，fractal	カメfractal, †**merge sort**
難 枝分かれ 合流あり	†**fibonacci**, nCm	一般迷路

† 試験範囲

枝分かれの例 — 木‡の探索 —

右と上方向だけ進める迷路．$(r, c) = (3, 1)$ から始める
到達可能なマス $C(3, 1)$ は何マス?

```
0 ##########
1 #.##....##
2 #.##.#####
3 #@....####
4 ##########
  0123456789
```

Ascii-Maze.ipynb

$$C(r,c) = \begin{cases} 0 & (r,c)\text{が壁のとき} \\ \underbrace{1}_{\text{現在地}} + \underbrace{C(r,c+1)}_{\text{右}} + \underbrace{C(r-1,c)}_{\text{上}} & \text{それ以外} \end{cases}$$

‡ 位置をノード，接続関係を辺とするグラフで表現すると，木構造に対応

枝分かれを伴う再帰

いろいろな再帰

	関数	手続き
易 線形	sum(n) gcd(a,b), sqrt(x,a,b)	printnum(n)
枝分かれ 合流無し	木，fractal	カメfractal, †**merge sort**
難 枝分かれ 合流あり	†**fibonacci**, nCm	一般迷路

† 試験範囲

枝分かれの例 — 木‡の探索 —

```
0 ##########
1 #3##..7.##
2 #.##.#####
3 #@..1.####
4 ##########
  0123456789
```

右と上方向だけ進める迷路．$(r, c) = (3, 1)$ から始める
拾える**最高の宝** $T(3, 1)$ は?

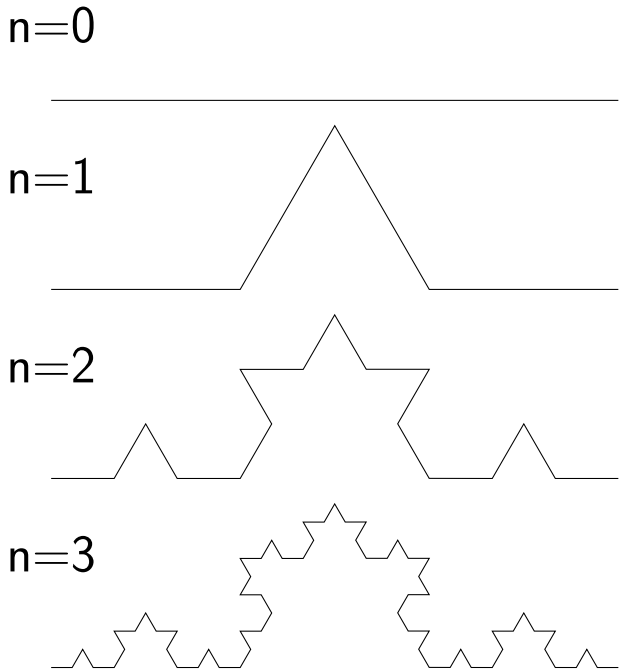
$$T(r,c) = \begin{cases} 0 & (r,c)\text{が壁のとき} \\ \max\left(\underbrace{n,}_{\text{現在地の宝}} \underbrace{T(r,c+1),}_{\text{右}} \underbrace{T(r-1,c)}_{\text{上}} \right) & \text{それ以外} \end{cases}$$

Ascii-Maze.ipynb

‡ 位置をノード，接続関係を辺とするグラフで表現すると，木構造に対応

Fractal — Koch curve

(鑑賞だけで良い)



実装の具体は数式の整理の手間が大きいので，試験範囲の学習が終わってからがお勧め

再帰構造の構成

$n \to n+1$ で各辺を 4分割． 各辺は相似．

作成方針

$K_n(s,t)$ $\doteq$ 点 s から点 t に n 段の Koch 曲線を描く各辺の始点の列†

$m_1, m_2, m_3 \doteq$ 辺を4分割する頂点a

$$K_n(s,t) = \begin{cases} [s] & n = 0 \\ K_{n-1}(s,m_1) + K_{n-1}(m_1,m_2) \\ \quad + K_{n-1}(m_2,m_3) + K_{n-1}(m_3,t)^{\ddagger} & n > 0 \end{cases}$$

† 重複防止: 終点=次の始点なので，sを含み，tを含まない

‡ ここの + 演算はリストの連結

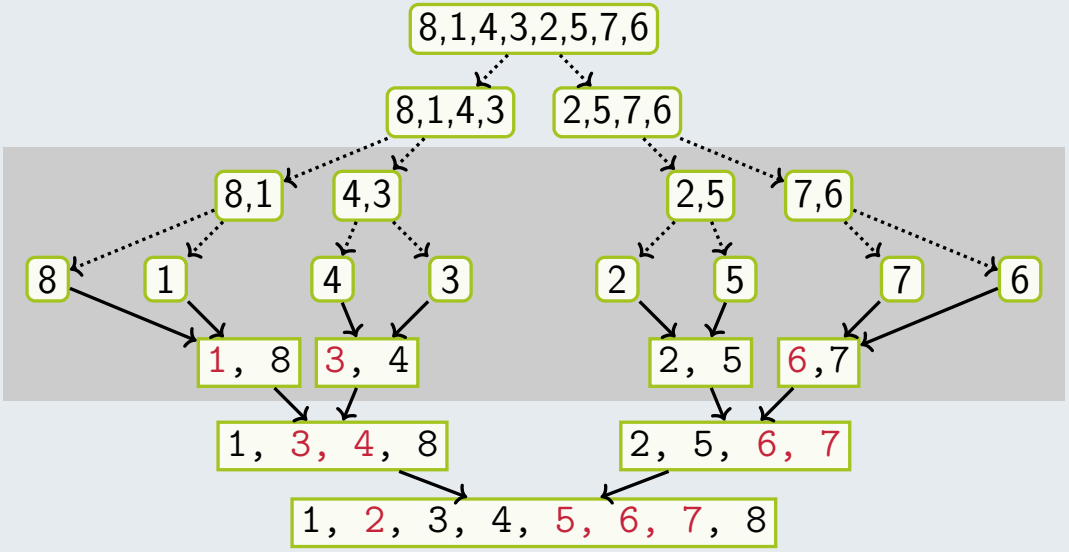
```
# 中点
cx, cy = (sx+tx)/2, (sy+ty)/2
# 進行方向
vx, vy = tx-sx, ty-sy
vx2, vy2 = -vy, vx # 90度左回転
c = math.sqrt(3)/6
```

```
# 中点から左手に c倍 伸ばす
m2x, m2y = cx+vx2*c, cy+vy2*c
# 1/3, 2/3 の内分点
m1x, m1y = (2*sx+tx)/3, (2*sy+ty)/3
# 2/3, 1/3 の内分点
m3x, m3y = (sx+2*tx)/3, (sy+2*ty)/3
```

Merge sort

教科書8.3, 試験範囲

再帰構造 — 左半分の整列と右半分の整列の合併




```
def mergesort(seq):
  if len(seq) <= 1: # basecase
    return seq
  half = len(seq)//2
  left = seq[0:half]
  right = seq[half:len(seq)]
  return merge(mergesort(left),
               mergesort(right))
```

新文法 **スライス** 配列[式左:式右]

seq=[8,1,4,3,2,5,7,6]

```
def mergesort(seq):
  ...
  half = len(seq)//2          # 4
  left = seq[0:half]          # [8,1,4,3]
  right = seq[half:len(seq)]  # [2,5,7,6]
  return merge(mergesort(left), mergesort(right))
                              # [1,3,4,8]  [2,5,6,7]
```

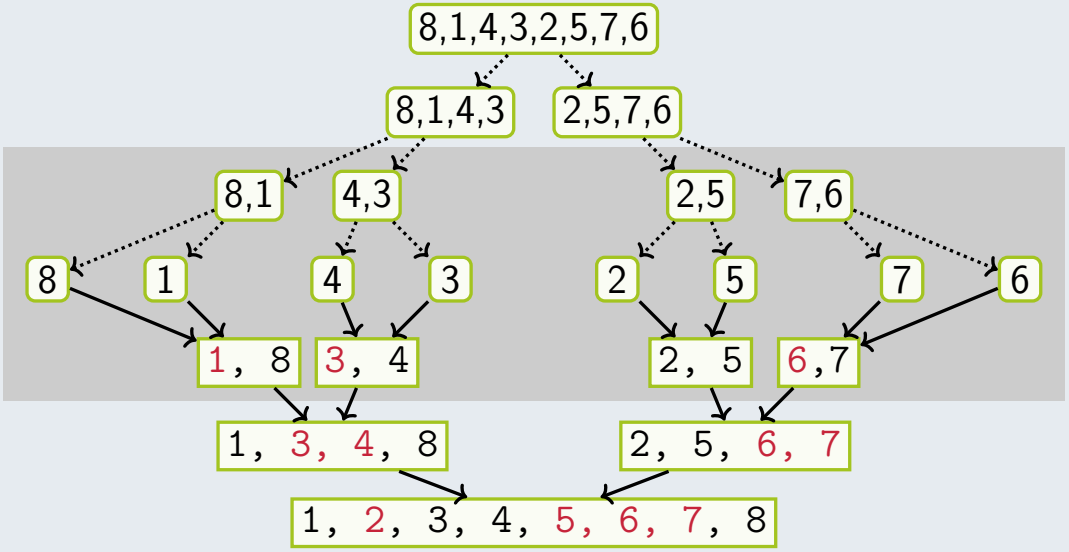
seq=[8,1,4,3]

```
def mergesort(seq):
  ...
  half = len(seq)//2          # 2
  left = seq[0:half]          # [8,1]
  right = seq[half:len(seq)]  # [4,3]
  return merge(mergesort(left), mergesort(right))
                              # [1,8]  [3,4]
```

Merge sort

教科書8.3, 試験範囲

再帰構造 — 左半分の整列と右半分の整列の合併




```
def mergesort(seq):
  if len(seq) <= 1: # basecase
    return seq
  half = len(seq)//2
  left = seq[0:half]
  right = seq[half:len(seq)]
  return merge(mergesort(left),
               mergesort(right))
```

新文法 **スライス** 配列[式左:式右]

- 実装 `merge(left, right)` があれば簡単
- 漸近計算量 $\mathcal{O}(N \log N)$　(配列長 N)
  水平に見ると各行で $\mathcal{O}(N)$ の処理，全体で $\mathcal{O}(\log N)$ 行

Merge 試作版


```
def merge(seq_a, seq_b): # 昇順の seq_a, seq_b を合併
  ans = ita.array.make1d(len(seq_a)+len(seq_b))
  ai = 0
  bi = 0
  # ループ不変条件
  # seq_a[ai] より前 seq_b[bi] より前の要素は ans[ai+bi] より前に昇順に格納済み
  # つぎに ans[ai+bi] に入る要素は seq_a[ai] と seq_b[bi]の小さい方
  while ai < len(seq_a) and bi < len(seq_b):
    if seq_a[ai] <= seq_b[bi]:
      ans[ai+bi] = seq_a[ai] #
      ai += 1
    else:
      ans[ai+bi] = seq_b[bi] #
      bi += 1
  # この時点で，片方だけ余っている   ai >= len(seq_a) または bi >= len(seq_b)
  while ai < len(seq_a):
    ans[ai+bi] = seq_a[ai] #
    ai += 1
  while bi < len(seq_b):
    ans[ai+bi] = seq_b[bi] #
    bi += 1
  return ans
```

seq_aから1つ移動

seq_bから1つ移動

seq_aから1つ移動

seq_bから1つ移動

Merge 教科書風

コードの重複部分を整理

while 統合版

```
ai, bi = 0, 0
while ai < len(seq_a) or bi < len(seq_b):
  if bi >= len(seq_b) or (ai < len(seq_a) and seq_a[ai] <= seq_b[bi]):
    ans[ai+bi] = seq_a[ai] #
    ai += 1
  else:
    ans[ai+bi] = seq_b[bi] #
    bi += 1
return ans
```

seq_aから1つ移動

seq_bから1つ移動

for 版

```
ai, bi = 0, 0
for ci in range(len(ans)):
  if bi >= len(seq_b) or (ai < len(seq_a) and seq_a[ai] <= seq_b[bi]):
    ans[ci] = seq_a[ai] #
    ai += 1
  else:
    ans[ci] = seq_b[bi] #
    bi += 1
return ans
```

seq_aから1つ移動

seq_bから1つ移動

論理演算子と短絡評価 short circuit

短絡評価 — 2項演算子のなかで and や or の特殊ルール

1. and や or の**左の項を先に評価**する
2. 左の項の真偽値で全体の真偽値が確定したら，右の項を評価しない
   確定しない場合のみ右の項を評価する

基本パターン　False and 何か ➔ False

`seq = [7,7,7]`, `len(seq) == 3` として

変数\式	`i<len(seq)`	`seq[i]==7`	`i<len(seq) and seq[i]==7`	`seq[i]==7 and i<len(seq)`
i=0	True	True	True	True
i=3	False	**エラー**	False	**エラー**
			(オススメ)	(非推奨)

Merge 教科書風

and の左右が重要

```
(ai < len(seq_a) and seq_a[ai] <= seq_b[bi])
```

授業としては読めれば良い.
自分で書く時は要注意

合流する再帰

いろいろな再帰

(再掲 ◀)

	関数	手続き
易 線形	sum(n), gcd(a,b), sqrt(x,a,b)	printnum(n)
枝分かれ 合流無し	木, fractal	カメfractal, †**merge sort**
難 枝分かれ 合流あり	†**fibonacci**, nCm	一般迷路

† 試験範囲

合流の困難 — 合流未対応のアルゴリズムは誤動作 —

正答
14

誤答
18 (合流地点！以降が，二重にカウントされる)

```
0 ##########
1 #...!...##
2 #.##.#####
3 #@....####
4 ##########
  0123456789
```

右と上方向だけ進める迷路．$(r,c)=(3,1)$ から始める
到達可能なマス $C(3,1)$ は何マス?

$$C(r,c)=\begin{cases}0 & (r,c)\text{が壁のとき} \\ \underbrace{1}_{\text{現在地}}+\underbrace{C(r,c+1)}_{\text{右}}+\underbrace{C(r-1,c)}_{\text{上}} & \text{それ以外}\end{cases}$$

!要修正!または訪問済み

合流する再帰

組合せ数 nCm

教科書7.3

$$ {}_nC_m \doteq \frac{n!}{(n-m)!m!} $$

$$ {}_nC_m = \begin{cases} 1 & m = 0 \quad \text{or} \quad m = n \\ {}_{n-1}C_{m-1} + {}_{n-1}C_m & 1 \leq m < n \end{cases} $$

```
def comb(n, m):
  assert m <= n
  if m == 0 or m == n:
    return 1
  return comb(n-1, m-1) +
  ↪  comb(n-1, m)
```

Fibonnacci数

教科書7.6

$$ F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} \end{cases} $$

```
def fib(n):
  if n <= 0:
    return 0
  elif n == 1:
    return 1
  return fib(n-1)+fib(n-2)
```

どちらも，簡単，正しい，動く，ただし**遅い**

➔ 簡単な工夫で速くなる (発展)

合流する再帰

組合せ数 nCm

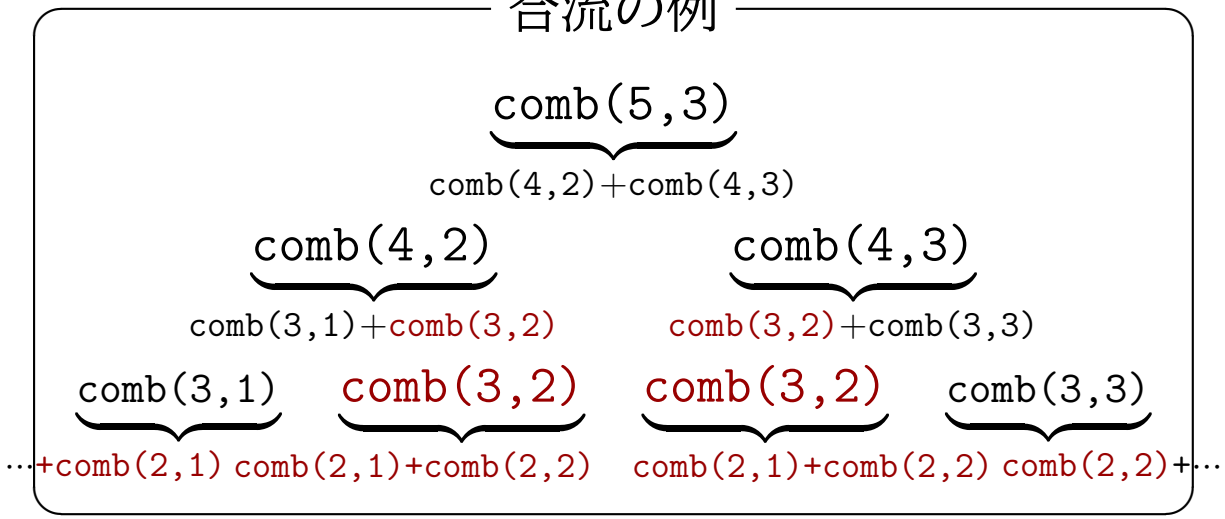
教科書7.3

$$ {}_nC_m \doteq \frac{n!}{(n-m)!m!} $$

$$ {}_nC_m = \begin{cases} 1 & m = 0 \quad \text{or} \quad m = n \\ {}_{n-1}C_{m-1} + {}_{n-1}C_m & 1 \leq m < n \end{cases} $$

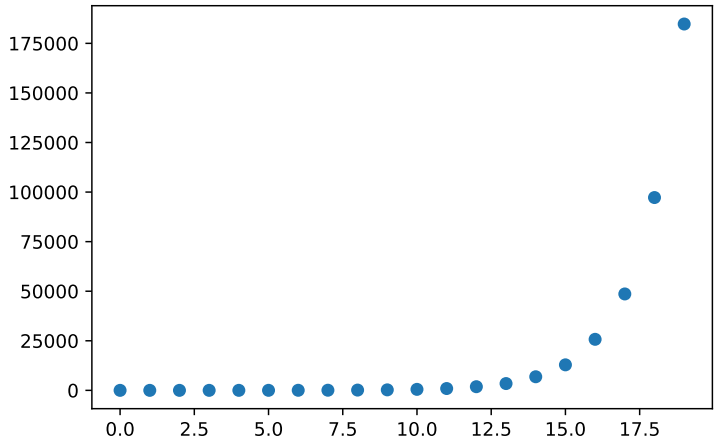
```
def comb(n, m):
  assert m <= n
  if m == 0 or m == n:
    return 1
  return comb(n-1, m-1) +
  ↪  comb(n-1, m)
```

合流の例



comb$\left(x, \frac{x}{2}\right)$ の再帰回数

(鑑賞で可)



合流する再帰と反復

[定型] 整数の漸化式は小さい方から計算 — c.f. 動的計画法 (発展)

Fibonnacci数 教科書7.6

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & \end{cases}$$

再帰版 (再掲)

```
def fib(n):
  if n <= 0:
    return 0
  elif n == 1:
    return 1
  return fib(n-1)+fib(n-2)
```

反復版

```
def fib_iter(n):
  table = ita.array.make1d(n+1)
    # 参考 table なしでも可能
  table[0] = 0
  table[1] = 1
  for i in range(2, n+1):
    table[i] = table[i-2]+table[i-1]
  return table[n]
```

合流する再帰と反復

[定型] 整数の漸化式は小さい方から計算 — c.f. 動的計画法 (発展)

組合せ数 nCm 教科書7.3

$$ {}_nC_m \doteq \frac{n!}{(n-m)!m!} $$

$$ {}_nC_m = \begin{cases} 1 & m = 0 \quad \text{or} \quad m = n \\ {}_{n-1}C_{m-1} + {}_{n-1}C_m & 1 \leq m < n \end{cases} $$

再帰版 (再掲)

```
def comb(n, m):
  assert m <= n
  if m == 0 or m == n:
    return 1
  return comb(n-1, m-1) +
  ↪  comb(n-1, m)
```

反復版パスカルの3角形

```
def pascal(n, m):
  assert m <= n
  table = ita.array.make2d(n+1,n+1)
  for i in range(n+1):
    for j in range(i+1):
      if j == 0 or i == j:
        table[i][j] = 1
      else:
        table[i][j] = table[i-1][j-1] +
        ↪  table[i-1][j]
  return table[n][m]
```

計算量 — 考え方 —

動機

複数の計算手順がある
➔ 良さを比較しよう

さまざまな良さの基準

- 正しさ — 最優先
- わかりやすさ
- 時間**計算量** — 早さ
- 空間計算量 — 省メモリ

計算量の測定方法候補

実時間 wallclock time

- pros. データ・環境固定なら最善
  ➔ 別紙で
- cons. 環境依存で再現性が難

「モデル」のステップ数

- pros. 客観的，抽象化
- cons. モデルと現実との整合 (乖離)

有名なモデル

- オートマトン ..... 簡素で分かりやすいが，Pythonを動かすには不十分
- チューリングマシン ..... P v.s. NP とか多項式時間 v.s. 指数時間の議論には有用
- Random access machine (RAM) ..... 現実に大分近いが，授業では難しい

➔ 妥協案「適当なモデル」で「**スケーラビリティ**」(scalability) を考えよう

スケーラビリティと計算量

ユースケース

小中規模の問題の処理時間を実測 ➔ 大規模な問題でも大丈夫 / ダメを予想

授業での手順

- 問題の大きくする部分を特定する
  e.g., 配列長，迷路の縦横
- N で表す
- N に依存するコストを集計する
- オーダで表現 e.g., $2N^2 + N + 3 \quad \sim \mathcal{O}(N^2)$

授業でのモデル

以下をコスト 1 単位と数える

- 式の演算子の評価
- 変数の読み書き
  演習 — 配列アクセス回数 `seq[i]`
- 文の実行
  (処理内容が N に依存しないもの)
- 関数の呼び出し
  (関数の内容の文は別にカウント)

一般的な前提

- 正しいアルゴリズムのみ対象 — 壊れて良いならいくらでも速くなる
- 大きい N に興味 — **漸近**計算量 — 細かい議論は難しい
- 同じNならコスト最大の入力を考える — **最悪**計算量 — c.f. 平均計算量

例 — 配列の長さ N に対するオーダ

整数の配列 seq (N=len(seq), $N \geq 1$) に対して，

先頭要素の3倍

```
def f(seq):
  return seq[0]*3 # コスト1
```

合計コストは N が増えても不変 ..... $\mathcal{O}(1)$

最小値

```
def minimum(seq):
  ans = seq[0] # コスト1
  for i in range(N):
    if seq[0] < ans: # 1×N回
      ans = seq[0] # 1×N回 (上限)
  return ans # コスト1
```

合計コストは $aN + b$ $(a, b \geq 0)$ ..... $\mathcal{O}(N)$

2番目の最小値の色々な実装

```
def sec_min_slow(seq):
  ans = seq[0] # コスト1
  for i in range(N):
    if min(seq) < seq[i] < ans:
       # コストN×N
       ans = seq[i] # コスト1×N
  return seq[1] # コスト1
```

(min(seq) の下: コストN)

..... $\mathcal{O}(N^2)$

```
def sec_min_unsafe(seq):
  seq.sort() # コスト O(N log N) 知識
  return seq[1] # コスト1
```

..... $\mathcal{O}(N \log N)$

for の中はループ回数分数える．if の中は上限回数を数える．

オーダ — 漸近記法 $\mathcal{O}$

(厳密な定義は試験範囲外)

$\mathcal{O}$記法 ~ 上界

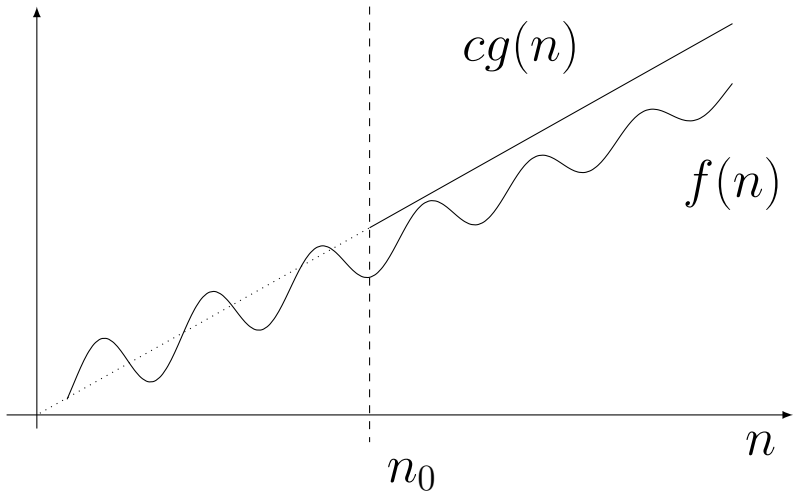
$$\underbrace{f(n)}_{\text{あるアルゴリズムの厳密な計算量}} \in \underbrace{\mathcal{O}(g(n))}_{\text{簡単な式 e.g., } n\log n, n^2}$$

$$\iff \underbrace{\exists c,}_{\text{定数は選べる}} \quad \underbrace{\exists n_0 > 0, \quad \forall n \geq n_0}_{\text{最初の有限個は例外があって良い}} \quad \underbrace{0 \leq \boldsymbol{f(n) \leq c \cdot g(n)}}_{\textbf{主文}}$$

理解の確認: 定義中の $\exists$ と $\forall$ の順番を入れ替えると全く異なる意味になる．この順番である必要性を分かりやすく説明せよ

使用例

- $n \in \mathcal{O}(n^2)$ 厳密には集合表記
- $n = \mathcal{O}(n^2)$ 略記，上界
- $3n = \mathcal{O}(0.001n)$
- $3x^3 + 8x^2 + 9 \notin \mathcal{O}(x)$
- $\log$ の底は関係ない
- 指数は大事 $3^n \notin \mathcal{O}(2^n)$



n は整数座標のみ考える

演習

道具 CountList

配列と同様の機能で要素の読み書きをカウントする (この授業専用)

```
>>> a = CountList([3,4,5]) # 作成
>>> a.access_count() # カウンタ読み取り
0
>>> a[0] # 要素を読んだら
3
>>> a.access_count() # 増える
1
>>> a[1] = 777 # 要素を変更しても
>>> a.access_count() # 増える
2
```

演習 — 性能調査

いろいろな入力 N に対する
グラフを描こう

unique_second_min @Part II

入力 seq，$N = \texttt{len(seq)}$

- N くらい — $\mathcal{O}(N)$ 最善
- $2N$ くらい — $\mathcal{O}(N)$
  さしあたり許容範囲
- N^2 くらい — $\mathcal{O}(N^2)$ 要改善

エラトステネスの篩 prime_array

友達の実装と比較してみよう.

時間測定

教科書

`ita.bench` .....使い方は試験範囲ではないので，この授業では割愛

実用 (試験範囲外)

- timeit 式 .....簡単なのでお勧め
  例リスト内包表記の速度

```
>>> %timeit [i**2 for i in range(1000)]
197 μs ± 1.24 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- cprofile .....授業範囲外, 少し大きなコードを書くようになったら
  `https://docs.python.org/ja/3/library/profile.html`

枝分かれする手続きの例 — n 進数の列挙 (鑑賞だけで良い)

2桁

```
def enum_twodigits(n): # 2桁
  for i in range(n):
    for j in range(n):
      print(str(i)+str(j))
```

```
>>> enum_twodigits(2) # 2進数2桁
00
01
10
11
```

```
>>> enum_twodigits(3) # 3進数2桁
00
01
02
10
11
12
20
...
```

3桁

```
def enum_threedigits(n): # 3桁
  for i in range(n):
    for j in range(n):
      for k in range(n):
        print(str(i)+str(j)+str(k))
```

```
>> enum_threedigits(2)
000
001
010
011
100
...
```

Q. m桁の列挙をするには?

m 重ループを書ければ良いが，実行時までは分からない ➔ 再帰

m桁 n進数の列挙 (鑑賞だけで良い)

m桁 n進数の列挙

```
def enum_digits_rec(prefix, m, n):
  for i in range(n):
    if m == 1:
      print(prefix + str(i))
    else:
      enum_digits_rec(prefix + str(i), m-1, n) # i の値ごとに枝分かれ
                      ~~~~~~~~~~~~~~~
                        次のprefix
def enum_digits(m, n):
  enum_digits_rec('', m, n)
```

2桁

```
def enum_twodigits(n): # 2桁
  for i in range(n):
    for j in range(n): ..... (*)
      print(str(i)+str(j))
            ~~~~~~
            prefix
```

3桁

```
def enum_threedigits(n): # 3桁
  for i in range(n):
    for j in range(n):
      for k in range(n): ..... (*)
        print(str(i)+str(j)+str(k))
              ~~~~~~~~~~~~
                prefix
```

Part IV に続く
(Topics)

Part IV

Topics

教科書のアルゴリズム

≈ 座学多

- ヒストグラムと`dict`
- 微分方程式の数値解法
- モンテカルロ法
- 数値誤差の理解
- 連立一次方程式

◂ 目次

特殊な文

通常の文

`print` のように， 1文づつ，順に実行される

特殊な文

- `def` — つづく関数定義ブロックは，定義に使われて，すぐには実行されない
- `return` — 関数から帰る
- `assert` — `False` なら，プログラム全体を終了
- `break` — new!
- `continue` — (試験範囲外，`break` の友達)
- `raise` — (試験範囲外，いわゆるエラーの仕組み)
- `yield` — (試験範囲外，Part II 発展で紹介した)

break

break

for または while を1レベル脱出する

例

整数の配列 seq について先頭から要素の合計を計算する．ただし，値 0 の要素があった場合は，それ以降を無視する

通常版

```
def sum_until_zero(seq):
  total = 0
  found_zero = False
  for i in range(len(seq)):
    if seq[i] == 0:
      found_zero = True
    elif not found_zero:
      total += seq[i]
  return total
```

break 版

```
def sum_until_zero_break(seq):
  total = 0
  for i in range(len(seq)):
    if seq[i] == 0:
      break # (*) この行が実行されると
    total += seq[i]
  # (**) この行に制御が移る
  return total
```

読めれば (書けなくて) 良い． 乱用すると読みにくくなり，バグの元．

early return

- 王道 — 基本パターンで，return は関数の最後．
  (今後もオススメ)
- 技解禁 — return はどこに書いても良い．
  (乱用するとバグの元)

関数の基本パターン for (再掲) ◂ 復習

```
def 関数名(引数1,...):
  変数 = 式     # 初期値
  for i in range(..):
    変数 = 式     # 値を更新
  return 変数 # return は基本最後
```

恣意的な例

```
def manyreturns():
  return 1 # early return 関数終了!
  return 2 # 以降は実行されない
  print('もしこの行が実行されたら')
  return 3
  print('結婚を申し込むんだ')
  return 4
```

```
>>> manyreturns()
1
```

エラー確認の例

```
def factorial(n):
  if n < 0:
    return 'error!' # early return
  ans = 1
  for i in range(n):
    ans *= i+1
  return ans
```

```
>>> factorial(3)
6
>>> factorial(-3)
'error!'
```

データの管理

ホテルの名簿管理

```
# おとくいさま名簿
name_list = ['Ieyasu', 'Hidetada', 'Iemitsu', 'Yoshimune']
```

実用 in, not in, index

```
>>> 'Hidetada' in name_list
True
>>> name_list.index('Hidetada')
2
>>> 'Maria' in name_list
False
>>> name_list.index('Maria')
Traceback (most recent call
↪  last):
  ...
ValueError: 'Maria' is not in
↪  list
```

自作 — 線形探索 linear search

```
def find_in_list(seq, key):
  for i in range(len(seq)):
    if seq[i] == key:
      return True # early return
  return False
```

```
>>> find_in_list(name_list,
↪  'Hidetada')
True
```

教科書 8.1

どちらの手法も $\mathcal{O}(N)$ (where N =len(seq)) ➔ 名簿が増えたら要工夫

ホテルの名簿管理つづき

```
# おとくいさま名簿
name_list = ['Ieyasu', 'Hidetada', 'Iemitsu', 'Yoshimune']
```

実用 set　　教科書では名前だけ登場 (8.5)

汎用の list に代えて，特化した set を使う

```
>>> names = set(name_list)
>>> 'Hidetada' in names
True
>>> 'Maria' in names
False
>>> names.add('Maria') # 新おとくいさま
>>> 'Maria' in names
True
```

速い (おおむね $\mathcal{O}(1)$, 技術は授業範囲外．「データ構造」を学ぶ授業で)

線形探索 linear search と二分探索 binary search

データが整列済みなら $\mathcal{O}(N)$ から $\mathcal{O}(\log N)$ に改善できる

例　当選番号リスト

線形探索 $\mathcal{O}(N)$ (再掲)

```
def find_in_list(seq, key):
  for i in range(len(seq)):
    if seq[i] == key:
      return True # early return
  return False
```

二分探索

アイデアは二分法 ◀ と共通

注意　書くのは高難易度

整数を扱う二分探索は，± 1の違いが誤動作に直結．安全に実装するなら，範囲を 10 程度まで絞って線形探索に切り替えても良い (漸近計算量は同じ)．

二分探索　教科書風 $\mathcal{O}(\log N)$

```
def binary_search(seq, key):
  left = 0
  right = len(seq)
  while left < right:
    c = (left + right) // 2
    if seq[c] == key:
      return True # early return
    if key < seq[c]:
      right = c # [left, c]
    else:
      left = c + 1 # [c+1, left]
  return False
```

† 超重要!

† +1 を忘れると無限ループ
[left,right] == [0,1] のとき c == 0 のため

線形探索 linear search と二分探索 binary search

データが整列済みなら $\mathcal{O}(N)$ から $\mathcal{O}(\log N)$ に改善できる

例　当選番号リスト

線形探索 $\mathcal{O}(N)$ (再掲)

```
def find_in_list(seq, key):
  for i in range(len(seq)):
    if seq[i] == key:
      return True # early return
  return False
```

二分探索

アイデアは二分法 ◀ と共通

注意　書くのは高難易度

整数を扱う二分探索は，± 1の違いが誤動作に直結．安全に実装するなら，範囲を 10 程度まで絞って線形探索に切り替えても良い (漸近計算量は同じ)．

二分探索　別解 $\mathcal{O}(\log N)$

```
def binary_search2(seq, key):
  left = 0
  right = len(seq)
  # 不変条件 key があるなら
  # 半開区間 [left, right) 内
  while left + 1 < right:
                †超重要!
    c = (left + right) // 2
    if key < seq[c]:
      right = c
    else:
      left = c
  # この時点で left + 1 == right
  return seq[left] == key
```

† +1 を忘れると無限ループ

データの管理 ⟨*key*, *value*⟩

dict　教科書 8.5

⟨*key*, *value*⟩ の集合の管理.
辞書，連想配列 (配列の添字を整数以外に拡張)

作成

- {}
- {key:value}
- {key1:value1, key2:value2, ...}

検索 ➔ bool

```
key in dict
```

value 取得

```
dict[key]
```

取得時に，key が未登録だとエラー

value 設定

```
dict[key] = value
```

設定は，key が未登録でも可

例　来月の予約リストの予約管理

リスト　検索 $\mathcal{O}(N)$　N =len(name_book_list)

```
name_book_list = [['Pers', 7], ['Herm',
↪  13], ['Orph', 2], ['Athen', 22]]
```

dict　検索おおむね$\mathcal{O}(1)$

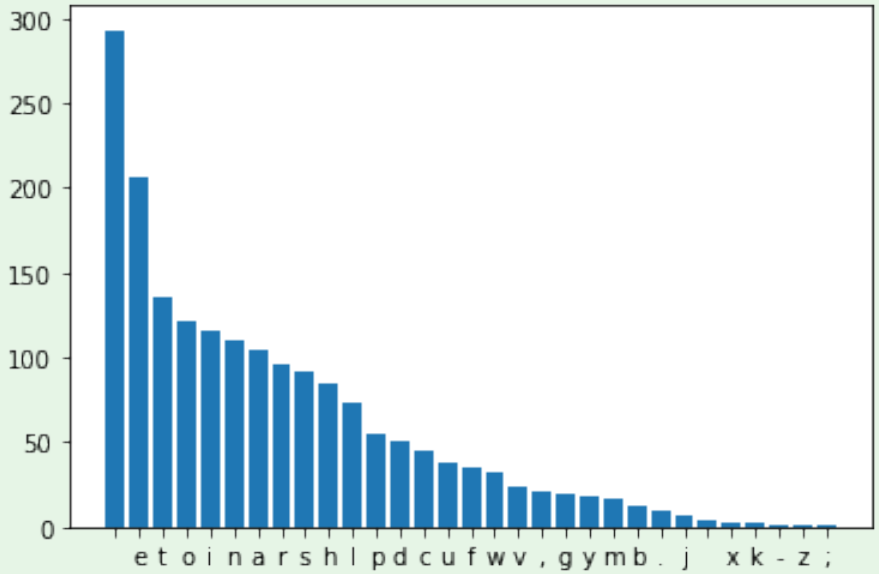
```
>>> name_book_dict = {'Pers': 7,
↪  'Herm':13, 'Orph':2, 'Athen':22}
>>> 'Athen' in name_book_dict # 便利
True
>>> name_book_dict['Athen']
22
>>> 'Izanagi' in name_book_dict
False
>>> name_book_dict['Izanagi']
Traceback (most recent call last):
 1 name_book_dict['Izanagi']
KeyError: 'Izanagi'
>>> name_book_dict['Izanagi'] = 30
>>> name_book_dict['Izanagi']
30
```

ヒストグラム key=文字, value=頻度

アルファベットの出現頻度

例文

We, the Japanese people, acting through our duly elected representatives in the National Diet, determined that we shall secure for ourselves and our posterity ...



こういうグラフを描くために，データを集計したい

教科書 8.5章風

```
def count_char(msg):
  freq = {} # dict
  for i in range(len(msg)):
    c = msg[i].lower() # 小文字に統一
    if c in freq:
      freq[c] += 1
    else:
      freq[c] = 1
  return freq
```

```
>>> count_char(cj)
{'w': 32,
 'e': 206,
 ',': 21,
 ' ': 293,
...}
```

出現頻度で sort したい ➔ 8.6章 (試験範囲外)

2変数のオーダ $\mathcal{O}$

(厳密な定義は試験範囲外)

ヒストグラム作成の計算量 (8.2章)　アイテム数 n カテゴリ数 m

リスト $\mathcal{O}(mn)$　v.s.　ソート済み あるいは dict $\mathcal{O}(n)$

$\mathcal{O}$記法 (1変数，再掲 ◀)

$$\underbrace{f(n)}_{\text{あるアルゴリズムの厳密な計算量}} \in \underbrace{\mathcal{O}(\underbrace{g(n)}_{\text{簡単な式 e.g., } n\log n,\, n^2})}$$

$$\iff \underbrace{\exists c,}_{\text{定数は選べる}} \quad \underbrace{\exists n_0 > 0, \quad \forall n \geq n_0}_{\text{最初の有限個は例外があって良い}} \quad \underbrace{0 \leq f(n) \leq c \cdot g(n)}_{\text{主文}}$$

$\mathcal{O}$記法 (2変数，**New!**)

$$\underbrace{f(\boldsymbol{m}, n)}_{\text{あるアルゴリズムの厳密な計算量}} \in \mathcal{O}(\underbrace{g(\boldsymbol{m}, n)}_{\text{簡単な式 e.g., } m^2+mn})$$

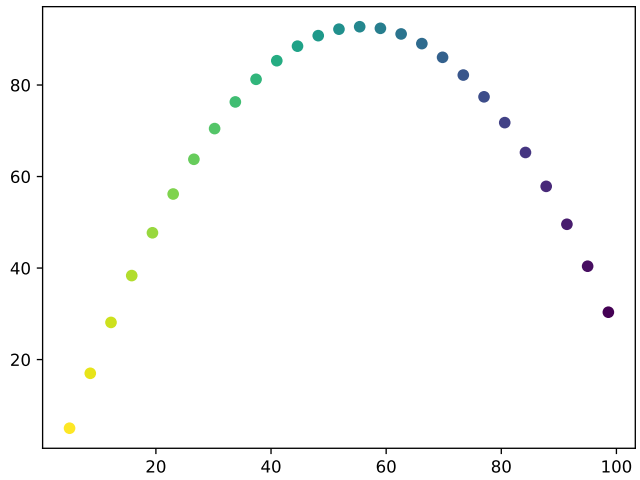
$$\iff$$

$$\underbrace{\exists c,}_{\text{定数は選べる}} \quad \underbrace{\exists n_0, \boldsymbol{m_0} > 0, \quad \forall (\boldsymbol{m}, n)\ \boldsymbol{m} \geq \boldsymbol{m_0}\ \textbf{or}\ n \geq n_0}_{\textbf{原点付近の長方形内の格子点}\text{は例外として良い}} \quad \underbrace{0 \leq f(\boldsymbol{m}, n) \leq c \cdot g(\boldsymbol{m}, n)}_{\text{主文}}$$

i.e., $\mathcal{O}(m^2+n)$, $\mathcal{O}(nm^2)$, $\mathcal{O}(m^2)$ などは区別して扱う．たとえば $3m^2+2n \notin \mathcal{O}(m^2)$

放物運動のシミュレーション~ 漸化式の計算 教科書6章

放物運動



時刻 t での位置 $(x(t), y(t))$，速度 $(v_x(t), v_y(t))$ 加速度 (a_x, a_y) は定数.

```
def step(x, y, vx, vy, ax, ay, stride):
  # 次の時刻の位置と速度を計算
  vx_next = vx + ax * stride
  vy_next = vy + ay * stride
  x_next = x + vx * stride
  y_next = y + vy * stride
  return [x_next, y_next, vx_next, vy_next]
```

初期位置 $(x(0), y(0)) = (5, 5)$, 初期速度 $(v_x(0), vy(0)) = (12, 40)$, 加速度 $(a_x, a_y) = (0, -9.8)$，stride $= 0.3$ とすると上記のグラフを得る.

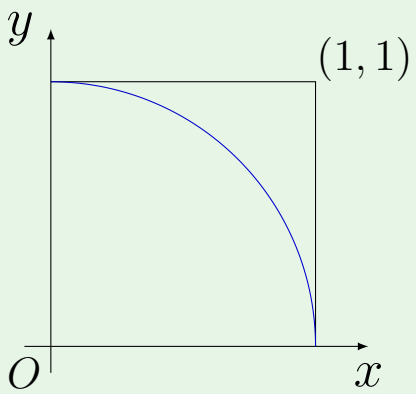
意味

放物運動は微分方程式を解いて厳密解を得られるが，解くことが難しい微分方程式も多い．そのような場合も，数値計算で近似解を得られる．この例は，数値計算のもっとも単純な手法．別スライド math.pdf も参照

モンテカルロ法

疑似乱数を使って，厳密な計算が難しい量を推定することができる
例題として知っている量を推定する

円周率の推定



x, y それぞれを $[0, 1]$ の一様分布からサンプルすると，図の正方形内のランダムな点に相当.
サンプル点数 N を増やすと，円の内側点の比率は面積と比例し，$1/4\pi$ 程度になるはず.
点が円の内外か判定できるが，面積の計算方法は分からないという立場.

```
>>> import random
>>> random.random() # [0,1] の乱数
0.561743836024374
>>> random.random() # 実行毎に変化
0.972277725313195
```

```
def inside(x, y):
  return x**2+y**2 < 1
def random_point():
  return [random.random(), random.random()]
```

```
>>> random_point()
[0.41734426724485885, 0.030401373447395574]
>>> random_point()
[0.7197547389737317, 0.6579844106881754]
```

数値誤差と要因

	整数	実数
精度 (刻み幅)	1	< 1
有限bit表現の影響	表現範囲	表現範囲+精度
加減算と誤差	不変	**拡大 するかも**

その他　打ち切り誤差

`math.sin`, `math.exp` などは本質的に近似計算しかできない.

今の教科書では言及ない(?) 名前の由来は，Taylor 展開の打ち切り.

有限bit数由来

丸め誤差

有限精度表現による誤差

e.g., 0.1は2進表現で循環小数

◂ 復習

有効桁数　誤差の評価

上から何桁正しいか

	近似例	誤差	有効桁数
π	3.1	± 0.05	2桁
$\times 10$	31.0	± 0.5	2桁

「浮動小数点」と関係

加算や減算で誤差が拡大 **New!**　要**直観の調整**

桁落ち　正の数の減算，負の数の加算

近い数の差分をとると有効bit (桁) 数が減少

$$\underbrace{1.231}_{\text{4桁}}\underbrace{\square\square\square}_{\text{誤差}} - \underbrace{1.230}_{\text{4桁}}\underbrace{\square\square\square}_{\text{誤差}} = \underbrace{0.001}_{\textbf{1桁!}}\underbrace{\square\square\square}_{\text{誤差}}$$

(その後の桁移動 e.g., $\times 100$ で誤差拡大)

情報落ち

桁の大きく異なる加算や減算で，小さな方の数の有効bitの一部またはすべてが消失

$$10^{20} \pm 1 \approx 10^{20}$$

注　全体の有効bit数は不変 (大きな数のものが保存)

固定小数点 v.s. 浮動小数点数

暗記不要

数のいろいろな表現方法のアイデア

10進数，abcdeの5桁の記憶場所があるとする

表現1 (符号無し整数)

普通の10進数: abcde
範囲 $[00000, 99999]$ を1刻み

表現2 (符号付き整数の1例)

ずらした10進数: $\mathtt{abcde} - 50000$
範囲 $[-50000, 49999]$ を1刻み

表現3 (固定小数点数の1例)

10進数を線形変換:
$(\mathtt{abcde} - 50000)/100$
範囲 $[-500.00, 499.99]$ を0.01刻み

表現4: 浮動小数点数(1)

$\mathtt{a.bc} \cdot 10^{\mathtt{de}}$ (**仮数部** a.bc, **指数部** de)
範囲 $[0, 9.99 \cdot 10^{99}]$ (広がった!)
刻み $[0, 0.01]$, $[9.98 \cdot 10^{99}, 9.99 \cdot 10^{99}]$ は..
範囲拡大の代償は刻み幅

表現5: 浮動小数点数(2)

$\mathtt{a.bc} \cdot 10^{\mathtt{de}-50}$
範囲 $[0, 9.99 \cdot 10^{49}]$
最小値 0 の次は $0.01 \cdot 10^{-50}$

標準的浮動小数
$\sim$ 表現5 + 2進表現 + 符号bit

浮動小数点数の実際

試験範囲外，教科書9章コラム

Python の浮動小数

```
>>> import math
>>> math.frexp(0.09375)
(0.75, -3)
>>> math.frexp(123.0)
(0.9609375, 7)
```

数 $x = M \cdot 2^E$ と分解．仮数部 M, 指数部 E

10進	指数表記	2進	指数表記‡	M (10進)	E (10進)
0.09375	$9.375 \cdot 10^{-2}$	0.00011	$0.11 \cdot 2^{-3}$	0.75	-3
123	$1.23 \cdot 10^{2}$	1111011	$0.1111011 \cdot 2^{7}$	0.9609375	7

‡ 小数点を最初の1の前に置く流儀．他では，最初の1のあとに小数点を置くので注意．

IEEE 754 倍精度 double (64bit)

教科書6.4コラム，試験範囲外

数 $x = \langle s, e, m \rangle$

3つの非負整数で表現

$$x = (-1)^s \underbrace{(1 + m \cdot 2^{-52})}_{\text{仮数}} \cdot 2^{\overbrace{(e - 1023)}^{\text{指数}}}$$

	位置	意味
符号部s	0	± 1
指数部e	1–11	小数点位置
仮数部m	12–63	数

観察

- 仮数の1番左の 1 を節約 0以外は必ずあるから
- 53bit 以内の整数は誤差なく表現可能
- 2進小数で53桁までは誤差なく表現可能†
  10進15桁程度

† 循環小数で丸め誤差がおきても，それなりに小さい．他の誤差が重要．

浮動小数点数の実際

試験範囲外，教科書9章コラム

Python の浮動小数

```
>>> import math
>>> math.frexp(0.09375)
(0.75, -3)
>>> math.frexp(123.0)
(0.9609375, 7)
```

数 $x = M \cdot 2^E$ と分解．仮数部 M, 指数部 E

10進	指数表記	2進	指数表記‡	M (10進)	E (10進)
0.09375	$9.375 \cdot 10^{-2}$	0.00011	$0.11 \cdot 2^{-3}$	0.75	-3
123	$1.23 \cdot 10^{2}$	1111011	$0.1111011 \cdot 2^{7}$	0.9609375	7

‡ 小数点を最初の1の前に置く流儀．他では，最初の1のあとに小数点を置くので注意．

IEEE 754 倍精度 double (64bit)

教科書6.4コラム，試験範囲外

数 $x = \langle s, e, m \rangle$

3つの非負整数で表現

$$x = (-1)^s \underbrace{(1 + m \cdot 2^{-52})}_{\text{仮数}} \cdot 2^{\overbrace{(e - 1023)}^{\text{指数}}}$$

	位置	意味
符号部s	0	± 1
指数部e	1–11	小数点位置
仮数部m	12–63	数

桁落ちとの対応

同じ e で m の上位ビットが揃った，異なる数で発生．

浮動小数点数の実際

試験範囲外，教科書9章コラム

Python の浮動小数

```
>>> import math
>>> math.frexp(0.09375)
(0.75, -3)
>>> math.frexp(123.0)
(0.9609375, 7)
```

数 $x = M \cdot 2^E$ と分解．仮数部 M, 指数部 E

10進	指数表記	2進	指数表記‡	M (10進)	E (10進)
0.09375	$9.375 \cdot 10^{-2}$	0.00011	$0.11 \cdot 2^{-3}$	0.75	-3
123	$1.23 \cdot 10^{2}$	1111011	$0.1111011 \cdot 2^{7}$	0.9609375	7

‡ 小数点を最初の1の前に置く流儀．他では，最初の1のあとに小数点を置くので注意．

IEEE 754 倍精度 double (64bit)

教科書6.4コラム, 試験範囲外

数 $x = \langle s, e, m \rangle$
3つの非負整数で表現

$$x = (-1)^s \underbrace{(1 + m \cdot 2^{-52})}_{\text{仮数}} \cdot 2^{\overbrace{(e - 1023)}^{\text{指数}}}$$

	位置	意味
符号部s	0	± 1
指数部e	1–11	小数点位置
仮数部m	12–63	数

情報落ちとの対応

2進 52 桁以上差がある数の加減算で顕著 (mの表現範囲外)

```
>>> 2**(-53) == 0 # 小さくても0ではないのだが
False
>>> 1 + 2**(-53) == 1.0 # 足しても変化しない
True
```

ガウスの消去法

連立一次方程式

$$\begin{aligned} 8x_0 + 3x_1 + 4x_2 &= 26 \\ 1x_0 + 5x_1 + 9x_2 &= 38 \\ 6x_0 + 7x_1 + 2x_2 &= 26 \end{aligned}$$

拡大係数行列

$$\begin{pmatrix} 8 & 3 & 4 & 26 \\ 1 & 5 & 9 & 38 \\ 6 & 7 & 2 & 26 \end{pmatrix} \doteq \begin{pmatrix} \boldsymbol{r}_0 \\ \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \end{pmatrix}$$

解が不変の操作 (例)

- 行の定数倍
- 行の入れ替え
- 行の，別の行への加算

▶ step 1

$$\left(\begin{array}{cccc} 1. & 0.375 & 0.5 & 3.25 \\ \hdashline 1. & 5. & 9. & 38. \\ 6. & 7. & 2. & 26. \end{array}\right) \quad 1/8$$

▶ step 2

$$\left(\begin{array}{cccc} 1. & 0.375 & 0.5 & 3.25 \\ 0. & 4.625 & 8.5 & 34.75 \\ \hdashline 6. & 7. & 2. & 26. \end{array}\right) \quad -\boldsymbol{r}_0^{\text{step1}}$$

▶ step 3

$$\left(\begin{array}{cccc} 1. & 0.375 & 0.5 & 3.25 \\ 0. & 4.625 & 8.5 & 34.75 \\ \hdashline 0. & 4.75 & -1. & 6.5 \end{array}\right) \quad -6\boldsymbol{r}_0^{\text{step1}}$$

▶ step 4

$$\left(\begin{array}{cccc} 1. & 0.375 & 0.5 & 3.25 \\ 0. & 1. & 1.837 & 7.513 \\ \hdashline 0. & 4.75 & -1. & 6.5 \end{array}\right) \quad 1/4.625$$

▶ step 5

$$\left(\begin{array}{cccc} 1. & 0.375 & 0.5 & 3.25 \\ 0. & 1. & 1.837 & 7.513 \\ 0. & 0. & -9.729 & -29.189 \end{array}\right) \quad -4.75\boldsymbol{r}_1^{\text{step4}}$$

▶ step 6

$$\left(\begin{array}{cccc} 1. & 0.375 & 0.5 & 3.25 \\ 0. & 1. & 1.837 & 7.513 \\ 0. & 0. & 1. & 3. \end{array}\right) \quad 1/(-9.729)$$

目標達成 (左下部分 0)　e.g., $x_2 = 3$

ガウスの消去法で使う道具

ベクトルの配列による表現

```
>>> v1 = [2, 0, 0]
>>> v2 = [0, 0.5, 1]
```

この例の直感的理解　3次元空間の各軸方向の移動量

実用

```
>>> import numpy as np
>>> v1 = np.array([2, 0, 0])
>>> v2 = np.array([0, 0.5, 1])
```

2つのベクトルの和

```
def vector_add(v1, v2):
  assert len(v1) == len(v2)
  ans = ita.array.make1d(len(v1))
  for i in range(len(v1)):
    ans[i] = v1[i] + v2[i]
  return ans
```

```
>>> vector_add(v1, v2)
[2, 0.5, 1]
```

この例の直感的理解　2種類の移動の結果

ベクトルのスカラー倍

```
def vector_scale(v, c):
  ans = ita.array.make1d(len(v))
  for i in range(len(v)):
    ans[i] = c * v[i]
  return ans
```

```
>>> vector_scale(v1, 0.5)
[1.0, 0.0, 0.0]
>>> vector_scale(v2, -1)
[0, -0.5, -1]
```

この例の直感的理解　移動量の拡大縮小

Part V

Follw up

Follow up 各章の例題の解説など

◂ 目次

簡潔なコードに

熟達者でもしばしばある

式を少しづつ修正していると意図せず冗長に ➔ 見直して簡素化しよう

整数の式に関する冗長表現例

```
部分式 + 0
```

```
部分式 + 1 - 1
```

➔

等価で簡潔な表現

```
部分式
```

このパターンは，高校までで訓練されているはず

論理式に関する冗長さの例

```
return 部分式 and True
```

```
return 部分式 or False
```

```
return bool(部分式)
```

```
if 論理式:
  return True
else:
  return False
```

➔

等価で簡潔な表現

```
return 部分式
```

このパターンの訓練は?
部分式が `True` でも `False` でも，等価なことを，真偽値表を書いて確認

Best practice

型変換は，当面，数と文字列の変換のみ使う．

簡潔なコードに 2

変数の定義? (実は動く)

```
[x] = [3]
```

```
[a,b] = [3, 100]
```

→

基本パターン

```
x = 3
```

```
a = 3
b = 100
```

多重代入 (発展)

```
a, b = 3, 100
```

正しい Python 一通り習得するまで非推奨

Best practice

- 基本パターンをしばらくは踏襲
- 王道に忠実に — 藪ではなく舗装道路を進もう

関数を部品として利用　close_enough

部品

境界含まない

整数a, b, c が $a < b < c$ を満たすなら True，満たさないなら Falseを返す関数

```
def in_range_strict(a,b,c):
  return ..
```

境界含む

同 $a \leq b \leq c$ なら True，満たさないなら Falseを返す関数 (不等号に等号を含む)

```
def in_range(a, b, c):
  return ...
```

応用

x, y の差が Delta (以下 Dと略記) 以内なら True, そうでなければFalse を返す関数

```
def close_enough(x, y, D):
  return in_range(-D,x-y,D)
```

これは数? 文字列? 変数?

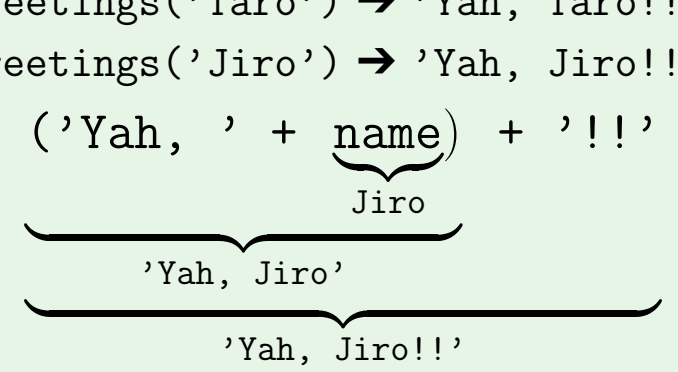
感想「greetings 難しかった」

```
def greetings(name):
  return 'Yah, ' + name + '!!'
```

難しさ: ここで name と書いて良いのか?

↑ **センスが良いかも**

- greetings('Taro') ➔ 'Yah, Taro!!'
- greetings('Jiro') ➔ 'Yah, Jiro!!'



理解

- 「文字列 + 文字列」は式.
- 変数の値が文字列なら「文字列 + 変数」は「文字列 + 文字列」と等価

解釈の規則

数	文字列	変数
057	'057'	-
-	'axb'	axb

1. '..' で囲まれる ➔ 固定文字列 (最優先)
2. 数字で始まる ➔ 数 (もしくはエラー)
3. 英字かアンダースコア _ で始まる ➔ 変数名

↑ **数と文字列は，対称ではない**

簡潔なコードに — in Part II —

熟達者でもしばしばある

コードを少しづつ修正していると意図せず冗長に ➔ 見直して簡素化しよう

if 文

```
if 条件1:
  文1
elif 条件2:
  文1
```

➔

等価で簡潔な表現

```
if 条件1 or 条件2:
  文1
```

例

```
if 条件1:
  a = a    # ?（非推奨）
else:
  a = 式
```

➔

等価で簡潔な表現

```
if not 条件1:
  a = 式
```

```
if 条件1:
  pass    # 何もしない文（試験範囲外）
else:
  a = 式
```

if の落とし穴

書くのは簡単だが，読むのは難しい
組み合わせるほど難解に

式と評価の補足 — Part I の内容の Part II での活用 —

加算における 0 っぽいもの

- 算術: `0 + x` ⟺ `x`
  e.g. `0 + 3` ⟺ `3`
- 文字列: `'' + x` ⟺ `x`
  e.g. `'' + 'Yo!'` ⟺ `'Yo!'`
- list: `[] + x` ⟺ `x`
  e.g. `[] + [7, 5]` ⟺ `[7, 5]`

他の演算

- 論理積: `True and x` ⟺ `x`
- 論理和: `False or x` ⟺ `x`

比較演算子追加

一致	（式 `==` 式）
不一致 (new!)	（式 `!=` 式）
	`not`（式 `==` 式）と等価

解説 is_increasing(a)

問題

整数の配列 a の要素が昇順なら True を，そうでないなら False を返す関数 is_increasing(a) を作成せよ．
len(a) $\geq$ 2 と仮定して良い
ここでは練習として if **文は使わずに** 実装する

入出力例

- [1,2] → True
- [1,1] → False
- [1,2,-3] → False

復習: こんなふうに書けるはず

関数の基本パターン (for)

```
def 関数名(引数1,...):
    変数 = 式        # 初期値
    for i in range(..):
        変数 = 式    # 値を更新
    return 変数
```

例: 0 から n-1 までの和

```
def sum_to_n(n):
    total = 0
    for i in range(n):
        total = total + i
    return total
```

解説 is_increasing(a)

問題

整数の配列 a の要素が昇順なら True を，そうでないなら False を返す関数 is_increasing(a) を作成せよ．
len(a) $\geq$ 2 と仮定して良い
ここでは練習として if **文は使わずに** 実装する

入出力例
- [1,2] → True
- [1,1] → False
- [1,2,-3] → False

2項演算で表現: $a_0 < a_1 < a_2 < a_3 \iff ((a_0 < a_1) \text{ and } (a_1 < a_2)) \text{ and } (a_2 < a_3)$

((0+1)+2)+3の場合

```
se = 0 # 空集合の和
s0 = se + 0 #  [0] の和
s1 = s0 + 1 #  [0, 1] の和
s2 = s1 + 2 #  [0, 1, 2] の和
s3 = s2 + 3 #  [0, 1, 2, 3] の和
```

$a_0 < a_1 < a_2 < a_3$ の場合

```
se = True # 空集合の論理積
s0 = se and (a0 < a1)
             真偽値
   # [a0, a1] の関係
s1 = s0 and (a1 < a2)
   # [a0, a1, a2] の関係
s2 = s1 and (a2 < a3)
   # [a0, a1, a2, a3] の関係
```

解説 is_increasing(a)

問題

整数の配列 a の要素が昇順なら True を，そうでないなら False を返す関数 is_increasing(a) を作成せよ．
len(a) $\geq$ 2 と仮定して良い
ここでは練習として if **文は使わずに** 実装する

入出力例
- [1,2] → True
- [1,1] → False
- [1,2,-3] → False

2項演算で表現: $a_0 < a_1 < a_2 < a_3 \iff ((a_0 < a_1) \text{ and } (a_1 < a_2)) \text{ and } (a_2 < a_3)$

増加判定 — 植木算注意 —

```
def is_increasing(a):
    ok = True
    n = len(a)
    for i in range(n-1):
        # (a)  a[0]..a[i] まで増加列
        ok = ok and (a[i]<a[i+1])
        # (b)  a[0]..a[i+1] まで増加列
    return ok
```

$a_0 < a_1 < a_2 < a_3$ の場合

```
se = True # 空集合の論理積
s0 = se and (a0 < a1)
    # [a0, a1] の関係
s1 = s0 and (a1 < a2)
    # [a0, a1, a2] の関係
s2 = s1 and (a2 < a3)
    # [a0, a1, a2, a3] の関係
```

植木算 — 要素が n 個 $\iff$ ペアは $n-1$ 個

List and index

添え字 (index) の有効範囲

リストを seq, 要素数を n=len(seq) として
- OK: seq[0] から seq[n-1] まで,
- NG: seq[n], seq[n+1], 以降全て

IndexError: list index out of range

「リストの添え字が範囲外 (多くは大きすぎ)」

off-by-one error: ± 1 は多くの bug のもと

例 seq = [0,1,2]
- len(seq) → 3
- seq[0] → 0
- seq[1] → 1
- seq[2] → 2
- seq[3] → IndexError

添え字 (index) の有効範囲 — 発展 —

リストを seq, 要素数を n=len(seq) として
- Pythonのみ: $\underbrace{\texttt{seq[-1]}}_{\to\texttt{seq[n-1]}}$ から $\underbrace{\texttt{seq[-n]}}_{\to\texttt{seq[n-n]}}$ まで
- NG: seq[-n-1], seq[-n-2], 以降全て

例2 seq = [0,1,2]

n = len(seq)
- seq[-1] → seq[n-1] → 2
- seq[-2] → seq[n-2] → 1
- seq[-3] → seq[n-3] → 0
- seq[-4] → IndexError

注意: Python以外のほとんどの言語で，負の添え字は不正

Loop and index

基本パターン: 全要素

```
def print_all(seq):
  for i in range(len(seq)):
    print(seq[i])
```

基本パターン: 全隣接ペア

```
def print_all_pair(seq):
  for i in range(len(seq)-1):
    print(seq[i], seq[i+1])
```

失敗例: 全隣接ペア

```
def print_all_pair(seq):
  for i in range(len(seq)):
    print(seq[i], seq[i+1])
```

→ `seq[len(seq)]`

.....i が増加して最後にはみだす

基本パターン: 和

```
def accumulate(seq):
  total = 0
  for i in range(len(seq)):
    total = total + seq[i]
  return total
```

Quiz1: 何が起こる?

```
def print_all_quiz(seq):
  for i in range(len(seq)):
    print(seq[i-1])
```

Quiz2: 何が起こる?

```
def accumulate_quiz(seq):
  total = 0
  for i in range(len(seq)):
    total = total + seq[i-1]
  return total
```

min_of と計算効率

関数の基本パターン (for+if)

```
def 関数名(引数1,...):
  変数 = 式        # 初期値
  for i in range(..):
    if 式:
      # (a) 更新 前 の i と変数の関係
      変数 = 式        # 値を更新
      # (b) 更新 後 の i と変数の関係
  return 変数
```

配列seq の要素の最小値

```
def min_of(seq):
  ans = seq[0]
  for i in range(len(seq)):
    #  (a) seq[max(0,i-1)]までの最小値
    if seq[i] < ans:
      ans = seq[i]
    #  (b) ans は seq[i]までの最小値
  return ans
```

計算のモデル (後日紹介)

- seq[i] の値の読み出しに，1単位かかる
- 比喩: seq ～ 本棚, seq[i] ～ i冊目の本を取り出す

上記のコードは，seq を1巡する．≈ まあまぁ効率が良い．
努力目標: 2番目の最小値 second_min なども，1巡ですませたい．(後回しでも良い)

解説 erase_ta(a)

問題

文字列 s から 'ta' を取り除いた文字列に相当する文字列を作成して返す関数 erase_ta(s) を作成せよ. 文字列 s は 2 文字以上. 'ta' の位置は，偶数文字目奇数文字目いずれもありうる.

入出力例

- 'otaru' → 'oru'
- 'ta' → ''
- 'at' → 'at'
- 'xyz' → 'xyz'

復習: こんなふうに書けるはず

関数の基本パターン (for+if)

```
def 関数名(引数1,...):
  変数 = 式      # 初期値
  for i in range(..):
    if 式:
      # (a) 更新 前 の i と変数の関係
      変数 = 式      # 値を更新
      # (b) 更新 後 の i と変数の関係
  return 変数
```

例: 0 から n-1 までの奇数の和

```
def odd_sum_to_n(n):
  total = 0
  for i in range(n):
    #  (a) 0..max(0,i-1)までの奇数の和
    if i % 2 == 1:
      total = total + i
    #  (b) total は 0..iまでの奇数の和
  return total
```

解説 erase_ta(a)

問題

文字列 s から 'ta' を取り除いた文字列に相当する文字列を作成して返す関数 erase_ta(s) を作成せよ. 文字列 s は 2 文字以上. 'ta' の位置は，偶数文字目奇数文字目いずれもありうる.

入出力例

- 'otaru' → 'oru'
- 'ta' → ''
- 'at' → 'at'
- 'xyz' → 'xyz'

惜しい習作

```
def erase_ta(s):
  ans = ''
  for i in range(len(s)):
    #  排除条件1: 右にa が続く t
    t_in_ta = (s[i] == 't' and
               s[i+1] == 'a')
    #  排除条件2: 左のt に続く a
    a_in_ta = (s[i-1] == 't' and
               s[i] == 'a')
    if not (t_in_ta or a_in_ta):
      ans = ans + s[i]
  return ans
```

添え字 i の範囲に注意

n = len(s) として

- i $\in$ range(n) $= [0, n-1]$
- i+1 $\in [1, n]$
  ➔ seq[n] は，IndexError
- i-1 $\in [-1, n-1]$
  ➔ seq[-1] は，右端の文字

解説 erase_ta(a)

問題

文字列 s から 'ta' を取り除いた文字列に相当する文字列を作成して返す関数 erase_ta(s) を作成せよ. 文字列 s は 2 文字以上. 'ta' の位置は，偶数文字目奇数文字目いずれもありうる.

入出力例

- 'otaru' → 'oru'
- 'ta' → ''
- 'at' → 'at'
- 'xyz' → 'xyz'

回答例1 条件を頑張る

```
def erase_ta(s):
  ans = ''
  for i in range(len(s)):
    t_in_ta = (i+1 < len(s) and
        s[i] == 't' and s[i+1] == 'a')
    a_in_ta = (i > 0 and
        s[i-1] == 't' and s[i] == 'a')
    if not (t_in_ta or a_in_ta):
      ans = ans + s[i]
  return ans
```

添え字 i の範囲に注意

n = len(s) として

- i $\in$ range(n) $= [0, n-1]$
- i+1 $\in [1, n]$
  ➔ seq[n] は，IndexError
- i-1 $\in [-1, n-1]$
  ➔ seq[-1] は，右端の文字

注「範囲ok and メインの式」の**左右の順序**が重要 (後日 ▸ p. 64)

解説 erase_ta(a)

問題

文字列 s から 'ta' を取り除いた文字列に相当する文字列を作成して返す関数 erase_ta(s) を作成せよ. 文字列 s は 2 文字以上. 'ta' の位置は，偶数文字目奇数文字目いずれもありうる.

入出力例

- 'otaru' → 'oru'
- 'ta' → ''
- 'at' → 'at'
- 'xyz' → 'xyz'

回答例2 両端を特別扱い

```
def erase_ta(s):
  ans = ''
  if not (s[0] == 't' and s[1] == 'a'):
    ans = s[0]
  for j in range(len(s)-2):
    i = j+1 # [1,..,len(s)-2]
    t_in_ta = s[i] == 't' and s[i+1] == 'a'
    a_in_ta = s[i-1] == 't' and s[i] == 'a'
    if not (t_in_ta or a_in_ta):
      ans = ans + s[i]
  if not (s[-2] == 't' and s[-1] == 'a'):
    ans = ans + s[-1]
  return ans
```

添え字 i の範囲に注意

n = len(s) として

- i $\in$ range(n) $= [0, n-1]$
- i+1 $\in [1, n]$
  ➔ seq[n] は，IndexError
- i-1 $\in [-1, n-1]$
  ➔ seq[-1] は，右端の文字

解説 erase_ta(a)

問題

文字列 s から 'ta' を取り除いた文字列に相当する文字列を作成して返す関数 erase_ta(s) を作成せよ. 文字列 s は 2 文字以上. 'ta' の位置は，偶数文字目奇数文字目いずれもありうる.

入出力例

- 'otaru' → 'oru'
- 'ta' → ''
- 'at' → 'at'
- 'xyz' → 'xyz'

回答例3 補助変数 + ループ不変条件

```
def erase_ta(s):
  ans = ''
  following_t = False
  for i in range(len(s)-1):
    t_in_ta = s[i] == 't' and s[i+1] == 'a'
    a_in_ta = following_t and s[i] == 'a'
    if not (t_in_ta or a_in_ta):
      ans = ans + s[i]
    # (a) 直前が 't' ➔ (b) 今が 't'
    following_t = (s[i] == 't')
  if not (following_t and s[-1] == 'a'):
    ans = ans + s[-1]
  return ans
```

添え字 i の範囲に注意

n = len(s) として

- i ∈ range(n) = $[0, n-1]$
- i+1 ∈ $[1, n]$
  ➔ seq[n] は，IndexError
- i-1 ∈ $[-1, n-1]$
  ➔ seq[-1] は，右端の文字

解説 unique_second_min(a)

問題

整数の配列 a の二番目の最小値を求める関数を作成せよ． a の要素に**重複はない**.
目標: a の要素を1巡読んで答えを得たい

入出力例

- [3,1,4,5] → 3
- [1,2,3] → 2
- [5,3,1,2] → 2

復習: こんなふうに書けるはず

関数の基本パターン (for+if)

```
def 関数名(引数1,...):
  変数 = 式      # 初期値
  for i in range(..):
    if 式:
      # (a) 更新 前 の i と変数の関係
      変数 = 式      # 値を更新
      # (b) 更新 後 の i と変数の関係
  return 変数
```

配列seq の要素の最小値

```
def min_of(seq):
  ans = seq[0]
  for i in range(len(seq)):
    # (a) seq[max(0,i-1)]までの最小値
    if seq[i] < ans:
      ans = seq[i]
    # (b) ans は seq[i]までの最小値
  return ans
```

解説 unique_second_min(a)

問題

整数の配列 a の二番目の最小値を求める関数を作成せよ． a の要素に**重複はない**.
目標: a の要素を1巡読んで答えを得たい

入出力例

- [3,1,4,5] → 3
- [1,2,3] → 2
- [5,3,1,2] → 2

答えは正しい

```
def uniq_second_min_slow(a):
  min1 = min(a) # 最小値（固定）
  min2 = max(a)
  for i in range(len(a)):
    # (a) min2 は最大値 (i==0)
    # (a) min2 > min1 で seq[i-1] までの最小
    if min1 <= a[i] < min2:
      min2 = a[i]
    # (b) min2 > min1 で seq[i] までの最小
  return min2
```

工夫

2つの変数 min1, min2 を用意

計算効率 (後日説明)

n = len(s) として
配列の要素を何回読むか?

- for i in range(i):
  ... a[i] ... 合計 n 回
- min(a) や max(a) で n 回

左の実装は，3n 回

解説 unique_second_min(a)

問題

整数の配列 a の二番目の最小値を求める関数を作成せよ． a の要素に**重複はない**.
目標: a の要素を1巡読んで答えを得たい

入出力例
- [3,1,4,5] → 3
- [1,2,3] → 2
- [5,3,1,2] → 2

回答例

```
def uniq_second_min(a):
  min1 = min(a[0], a[1])
  min2 = max(a[0], a[1])
  for j in range(len(a)-2):
    i = j+2 # i in [2,len(a)-1]
    # (a) min1 は seq[i-1] までの最小
    # (a) min2 は同範囲で次の最小
    if a[i] < min1:
      min2 = min1
      min1 = a[i]
    elif a[i] < min2:
      min2 = a[i]
    # (b) min1 は seq[i] までの最小
    # (b) min2 は同範囲で次の最小
  return min2
```

工夫

2つの変数 min1, min2 を用意

計算効率 (後日説明)

n = len(s) として
配列の要素を何回読むか?

- for i in range(i):
  ... a[i] ... 合計 n 回
- min(a) や max(a) で n 回

左の実装は，約 n 回

解説 second_min(a)

問題

整数の配列 a の二番目の最小値を求める関数を作成せよ． a の要素に**重複がありうる**.
目標: a の要素を1巡読んで答えを得たい

入出力例

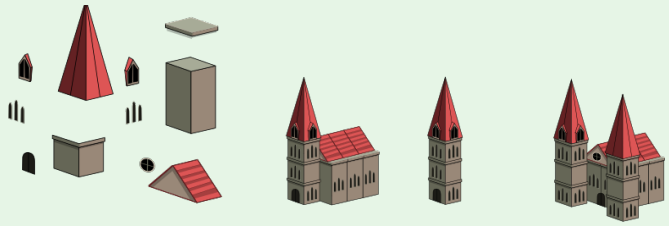
- [3,1,4,5] → 3
- [1,2,1,3] → 1
- [5,3,1,2] → 2

回答方針

uniq_second_min 同様に，処理範囲の最小値と次の最小値を更新する.

Part III followup

PDF目次



組み立ての心得 — Part2 では Part1の部品をいくつか組み立てる

エラーは誰でも (上級者でも) あるので淡々と対応

- 添え字の間違い　e.g, `i`, `j` が逆，± 1 ずれる
- 型の間違い　e.g., 配列のつもりが整数，整数のつもりが配列
- うっかり　e.g., インデント不揃い，コロン`:`忘れ，`range`忘れ，`len`忘れ

デバッグのヒント

- エラーメッセージを読む　`XXXError` の `XXX`毎にパターン化
- 長い式は，別のセルで部分式を評価
- エラー発生の直前の行で `print` 追加
- 別のセルで，似たような関数を作る．どの機能を加えるとエラーが起こるか?

小数の2進表現

2進表現

10進	2進
0	0
0.5	0.1
0.25	0.01
0.125	0.001
1/16	0.0001
0.75	0.11

練習 — 小数 x の小数点以下 n 桁の2進表現を得る関数を書いてみよう

実装例 (鑑賞だけで良い)

```
def mybinf(x, n):
  assert 0 <= x < 1 # 対応している対象を明示
  ans = '0.'
  for i in range(n):
    x = x*2
    if x >= 1: # 小数点以下i桁目が1 ⟺ 2^i倍が1
      ans = ans + '1'
      x = x - 1 # 整数部分を0に
    else:
      ans = ans + '0'
  return ans
```

よくある罠

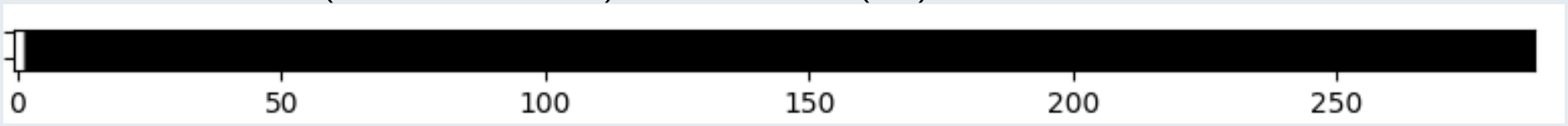
0.1 のような十進小数で構成しようとすると，微少な誤差が発生．文字列で作る．

エラトステネスの篩

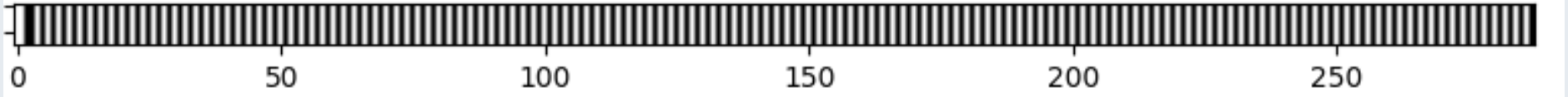
試験範囲外，教養として紹介

アルゴリズム概略

- 0, 1 を白にmark (素数ではない)．他は不明 (黒)

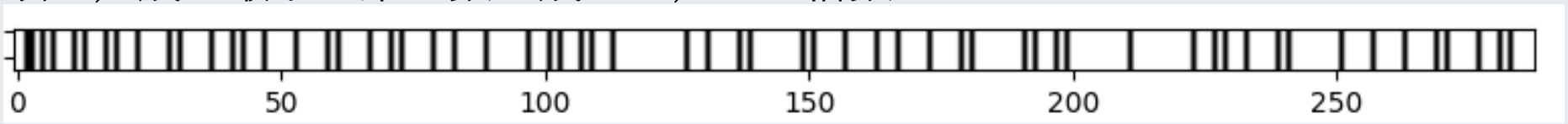


- 2を残して，2の倍数をmark



- 同様に，3を残して，3の倍数をmark

- 4は，mark済みなので，なにもせず5に進む
- 順に，残る最小の紫の数を残して，その倍数をmark



- 残った黒が素数

効率 — 大きな素数はまばらなので，mark する頻度も減ってゆく

エラトステネスの篩 — 実装

篩の1ステップ相当 `mark_multiples(seq, n)`

真偽値の配列 `seq` について `n` より大きな `n` の倍数 `i` について `seq[i]` を `True` とせよ．それ以外の要素は変更しない．

実装例1　1つづつ確認

```
def mark_multiples(seq, n):
  for i in range(len(seq)):
    if i % n == 0 and i > n:.... ‡
      seq[i] = True ............. †
```

実装例2　nとばしに処理

```
def mark_multiples(seq, n):
  i = 2*n
  while i < len(seq):
    seq[i] = True.................. †
    i += n
```

実装例3　実用的別解

```
>>> list(range(6, 20, 3))                    1
[6, 9, 12, 15, 18]                           2
```

どちらの実装が効率的?

➔ `n` が大きいと，実装例2の方が効率的

`N=len(seq)`として

- † の行　　`N/n` 回程度 (両方)
- ‡ の行　　`N` 回程度 (実装例1のみ存在)

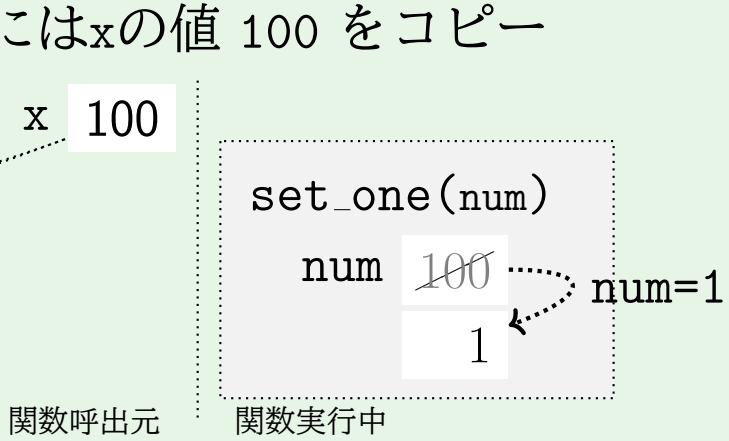
重箱の隅　フレームとローカル変数

- 「関数の呼び出し元の変数」と「関数の引数やlocal変数」は別のメモ帳
- 「参照先」(口座) を変更すると，同じ参照 (口座番号) を持つ複数変数から可視

重箱の隅　配列ではなく整数の値を1に設定してみよう

```
def set_one(num):
  num = 1
```

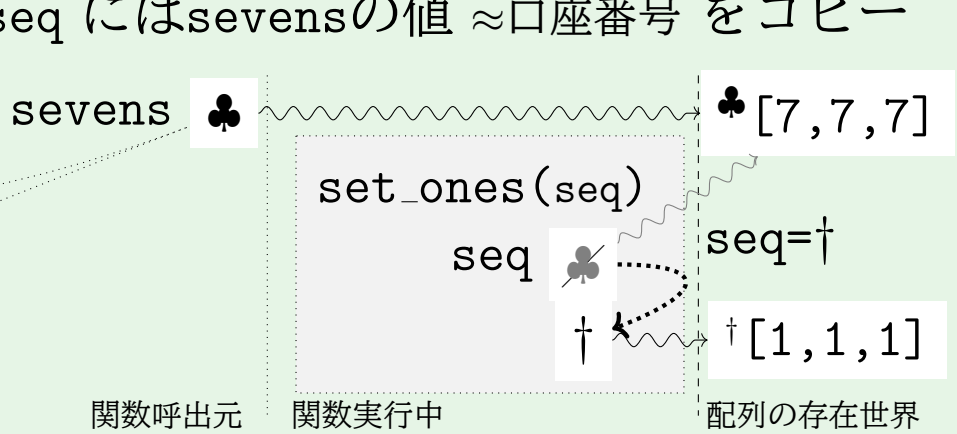
```
>>> x = 100
>>> set_one(x)
>>> x
100
```



重箱の隅2　いちいち要素を設定しなくてもまとめて設定すれば良いのでは?

```
def set_ones(seq):
  seq = ita.array.make1d(len(seq), 1)
```

```
>>> sevens = [7,7,7]
>>> set_ones(sevens)
>>> sevens
[7,7,7]
```



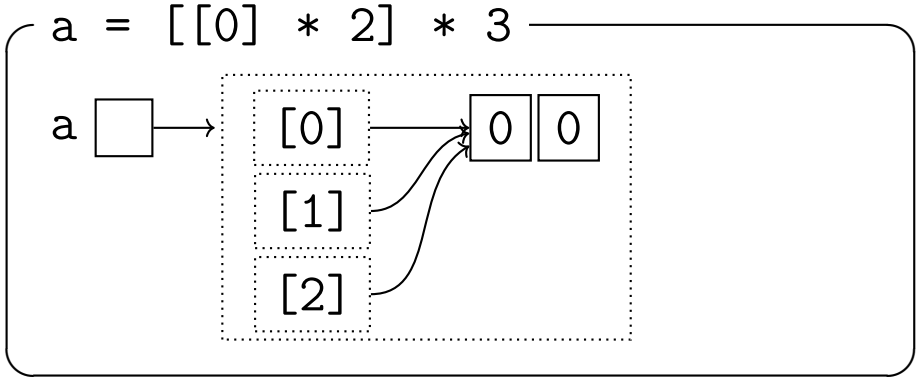
Q. なんでだめなの?

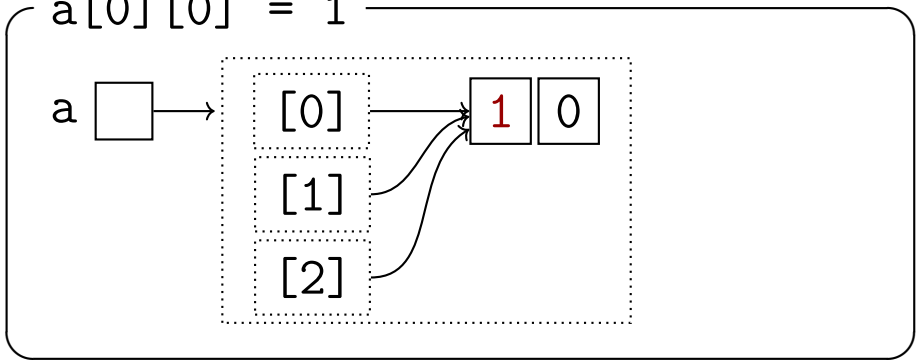
罠 — 似て非なるもの —

```
[[0] * r] * c # ダメ
```

ダメって言われても使ってみると...

```
>>> a = [[0] * 2] * 3
>>> a
[[0, 0], [0, 0], [0, 0]]
>>> show_matrix(a)
---------
   0   0
   0   0
   0   0
---------
>>> a[0][0] = 1
>>> show_matrix(a) # うん?
---------
   1   0
   1   0
   1   0
---------
```





おわり

目次