

プログラミング演習 (学際科学科)  
広域システム特論 III(3) (学際科学科)  
総合分析情報学基礎 IX (情報学環)

金子 知適 (kaneko@acm.org), 山口和紀

2020 年度 夏学期 (案) 再配布禁止

必ず授業当日に毎週ダウンロードし直すこと

Time-stamp: <2020-06-24 11:52:02 kaneko>

# 目次

第 1 章	ガイダンス・文の実行と制御構造	4
1.1	ガイダンス	4
1.2	環境整備	4
1.3	プログラムの作成と実行	7
1.4	順次の実行	7
1.5	整数と繰り返し	8
1.6	条件分岐	11
1.7	総合問題	11
1.8	この章の課題	13
1.A	毎週の課題の提出方式について	13
第 2 章	式と評価, 関数	16
2.1	式の評価	16
2.2	関数的な計算: 式と評価	17
2.3	参考: 命令的な計算	26
2.4	(発展) 関数を返す関数	27
2.5	この章の提出課題	31
2.A	ターミナルでの Python3 の利用	32
2.B	参考資料	33
第 3 章	ユニットテスト, 配列, ループ不変条件	34
3.1	テストの自動化: ユニットテスト	34
3.2	配列と関数	37
3.3	変数の値の書き換えとループ不変条件	39
3.4	配列とループ	42
3.5	配列や文字列の変更	43
3.6	関数のパラメータ化 (発展)	46
3.7	この章の提出課題	47
3.A	テスト駆動開発★	48
3.B	C++ でのテスト	51
第 4 章	参照と二次元配列	52

4.1	配列の要素の書き換え	52
4.2	二次元配列	54
4.3	有限ライフゲーム	57
4.4	実体と参照 *	61
4.5	この章の提出課題	65
4.A	先週の補足	66
第 5 章	再帰	67
5.1	帰納的定義と線形再帰	67
5.2	範囲を絞る: 二分法・二分探索	68
5.3	枝分かれを伴う再帰	72
5.4	併合整列 (Merge Sort)	76
5.5	再帰と反復の応用	78
5.6	この章の提出課題	80
5.A	関数と実行状態の管理	81
5.B	ユークリッドの互除法と応用	82
第 6 章	木構造, 複合データ (クラス)	84
6.1	階層的構造	84
6.2	クラスの利用の初歩	93
6.3	発展問題	94
6.4	この章の提出課題	95
第 7 章	状態管理, 抽象インターフェースと継承	97
7.1	内部状態を持つオブジェクト	97
7.2	インターフェース	107
7.3	この章の提出課題	113
第 8 章	クラスライブラリ・デザインパターン	114
8.1	クラスとオブジェクト	114
8.2	Python の help と docstring	118
8.3	クラスとインターフェースの継承	119
8.4	デザイン・パターン	123
8.5	C++ で class を扱う注意点	128
8.6	この章の提出課題	133
第 9 章	状態の保存と通信	134
9.1	事務連絡: 今後の予定	134
9.2	文字列とバイト列	134
9.3	テキストファイルの読み書き	135
9.4	dict (連想配列)	138

9.5	numpy . . . . .	141
9.6	ファイル実行 . . . . .	142
9.7	この章の提出課題 . . . . .	145
第 10 章	最終課題 . . . . .	146
10.1	最終課題の選択肢 . . . . .	146
10.2	提出期限 . . . . .	146
第 11 章	最終課題作成 . . . . .	148
第 12 章	最終課題発表 . . . . .	149
第 13 章	(栃木実習休講) . . . . .	150
付録 A	開発ツール . . . . .	151
A.1	バージョン管理システムと最小限の git . . . . .	151
A.2	ネットワーク経由の開発: SSH . . . . .	154
A.3	リモートリポジトリの利用と共同開発 . . . . .	155
A.4	処理の自動化 . . . . .	157
付録 B	最終課題: 潜水艦ゲーム . . . . .	159
B.1	目標 . . . . .	159
B.2	環境設定と対戦 . . . . .	159
B.3	作り方 . . . . .	161
付録 C	最終課題: OpenAI Gym . . . . .	162
C.1	インストール . . . . .	162
C.2	動作確認 . . . . .	162
C.3	ランダムプレイ . . . . .	163
C.4	賢い行動をさせる . . . . .	163
参考文献	. . . . .	165
索引	. . . . .	166

## 第 1 章

# ガイダンス・文の実行と制御構造

*Introduction, Statement and Control structure*  
2020-04-08 Wed

### 事務連絡

全学の方針により、今学期の授業はオンライン (Zoom) で行う。



### 環境に関する注意

本日の資料の 1.2.2 節は、情報教育棟の端末の利用を前提に書かれている。本年は状況から各自の PC を前提として授業を行うため、jupyter を各自でインストールする必要がある。以下の「Python 入門」の資料を参考にとすると良い。 [https://utokyo-ipp.github.io/0\\_guidance/guidance1.html](https://utokyo-ipp.github.io/0_guidance/guidance1.html) インストールに困難がある場合など、当面 (2 週間程度) は、1.2.4 節に紹介する Google Colaboratory を利用しても良い。その場合、提出方法の指示などの jupyter は、適宜 Google Colaboratory と読み替える。

## 1.1 ガイダンス

- 演習の形式・評価: 毎週の提出 + 最終課題
- 使用言語 Python3 (C++ は指定の一部課題のみ許可)
- 標準環境 jupyter-notebook

## 1.2 環境整備

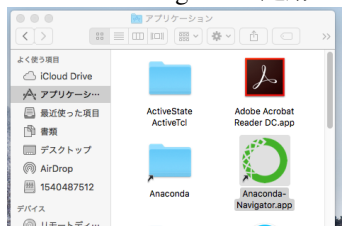
### 1.2.1 ITC-LMS への登録

水曜 5 限, プログラミング演習 (学際科学科, 広域システム特論 III(3) も可), 総合分析情報学基礎 IX (情報学環総合分析情報学コース) で登録する。「教材」の欄に本資料が PDF においてあるはずである。

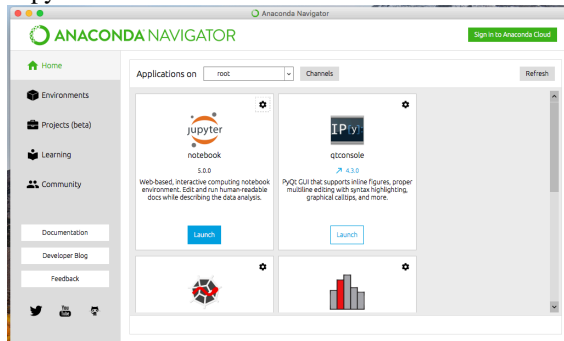
### 1.2.2 jupyter-notebook の起動と操作

情報教育棟の ECCS の iMac 端末を利用する場合

- アプリケーションフォルダを開く
- Anaconda-Navigator を起動



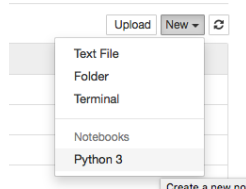
- Jupyter Notebook を起動



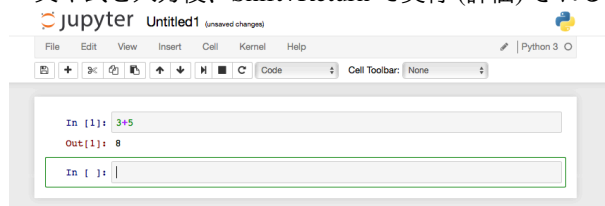
ブラウザにjupyter が起動する



右上の New というメニューから Python3 を選択



文や式を入力後、Shift+Return で実行 (評価) される



### 1.2.3 テンプレートから notebook を作成する

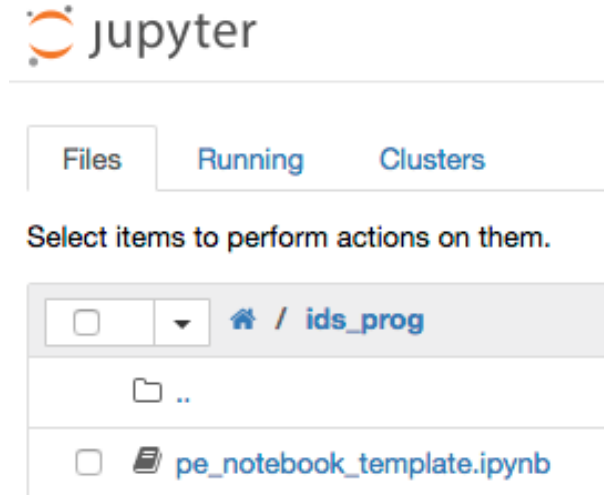
#### テンプレートの入手

ITC-LMS の本授業のページにアクセスし、「教材」に置いてある `pe.notebook.template.ipynb` をダウンロードする。提出用テンプレートを適当な場所 (例えば、ホームディレクトリ直下に `ids_prog` などの名前のディレクトリを作成する) へ移動する。

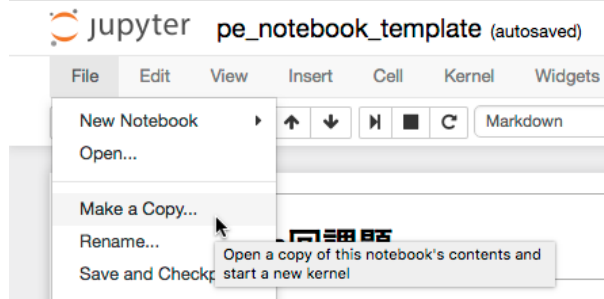
テンプレートから新しい notebook を作る

授業中にプログラムを実行したり作業するための作業用 notebook を作成する。

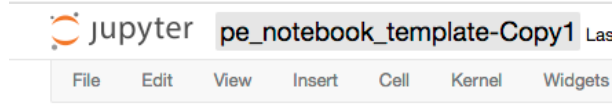
jupyter notebook 上でテンプレートファイルを開く。



File > Make a Copy でテンプレートファイルをコピーしたファイルを作成する。



画面上の pe\_notebook\_template-Copy1 をクリックし、新しいファイル名 (week0-work) へと変更する。



以降はここで作成したノートブック week0-work でプログラムを実行する。

#### 1.2.4 Google Colaboratory

jupyter のインストールに困難があった場合は、当面の代替手段として Google Colaboratory を使っても良い。ほぼ同様に動作すると思われる。ただし、トレードオフがあるので、両方使えることが望ましい。授業としては、早めに jupyter に移行することを推奨する。

<https://colab.research.google.com/>

## 1.3 プログラムの作成と実行

### 概要

プログラミングをどこから説明すると分かりやすいかは議論があるが、今年のこの演習では文の列から開始する:

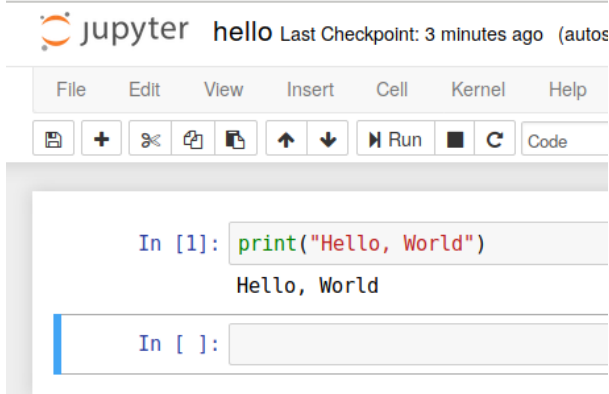
- プログラムは「文」(statement)の列から構成される
- 文は原則として前から順に実行される
- 「文」としては `print` (表示)のみを扱う。「実行」とは表示が行われることを意味するとする。
- 文の実行順序を変更する仕組みである制御構造を2つ導入する。ひとつは繰り返し `for`, もうひとつは条件分岐 `if` である

プログラムを構成する重要な概念である、式、変数、関数については次章以降に導入する。なるべく授業時間中に例題や問題に多く取り組んで、教員にチェックを受けること。

慣例に則り `Hello, world` と表示するプログラムを作成しよう。

```
Python3 1 print("Hello, world")
```

jupyter-notebook 上で上記を入力し、実行ボタンを押し、表示を確認する。



## 1.4 順次の実行

プログラムは「文」の列から構成される。この資料では(まずは)文として `print("xxx")` という形式の Python の表示 (`print`) 文を扱う。この文が実行されると二重引用符で囲まれた部分を表示するという効果を持つ。演習環境では、日本語など ASCII に含まれない文字も表示可能だが、文字コードを `utf-8` にしておくこと (<http://hwb.ecc.u-tokyo.ac.jp/wp/literacy/editor-2/emacs/encoding/>)。

```
Python3 1 print("春眠不觉晓")
        2 print("处处闻啼鸟")
        3 print("夜来风雨声")
        4 print("花落知多少")
```



改行文字は任意の場所に入れられるので、以下のように文との対応を変えても結果は同じである。また#から行末までは、コメントとして扱われ、実行に影響を与えない。さらに、**改行**を表す文字 `\n` を用いて、文字列の途中で改行を行うことができる。なお、改行文字を用いなくても `print` 文の終了時には改行が自動で行われる。この自動的な改行を抑制する手法は後で述べる。

```
Python3 1 print("春眠不覚暁")
        2 # ここで息継ぎ
        3 print("处处聞鳴鳥\n夜来風雨声\n花落知多少")
```

次の分は、改行を2回行う。

```
Python3 1 print("\n")
```

例題

改行 (newline)

(金子)

改行を適切に増やして、春暁の各行の間に空行が表示されるようにして、読みやすくせよ。

## 1.5 整数と繰り返し

文字列の他に 123 のような**整数**を扱うことにし、`print(123)` という表示を導入する。

以下のコードは各行に 1,2,3 の数字を一つずつ表示する。

```
Python3 1 print(1)
        2 print(2)
        3 print(3)
```

さて、3 までならば容易だったが、100 まで表示する場合に同様に記述すると 100 行も必要である。これは苦痛である<sup>\*1</sup>。そこで次のような `for` という略記法が用意されている。

```
Python3 1 for i in range(1, 4):
        2     print(i)
```

これは2行目の**制御変数** `i` の部分を 1,2,3 と次々と置き換えながら、`for` 文以降の一定の範囲を繰り返し実行する文法である。繰り返す範囲は**インデント**と呼ばれる行の先頭の空白文字で制御する。上記の2行目には、先頭に空白文字を4つ入れてある。python ではこれらの空白文字の有無がプログラムの動作に影響を与える。

`for` 制御変数 in range(始点, 終点):

繰り返し実行する文 (制御変数は、始点, 始点 +1, ..., 終点-1 に順に置き換わる)

今後、プログラムの意味を明確に保ったまま短く書くための概念や手法がしばしば登場する。必要があれば、計算機の気持ちになって実行をシミュレートする能力と、複雑な動作を簡潔に表現する能力の両方を磨い

<sup>\*1</sup> もしかすると数百行ならものともしないという人もいるかもしれないが、苦痛に思ってください

てほしい。

例題	100 (p100)	(金子)
1 から 100 まで表示せよ。		

例題	棒 (bar)	(金子)
<p>以下のような棒を表示せよ。以下の例では長さ 4 だが 70 など長いものも簡単な変更で書けるように作成し、実際に 70 で表示してみよ。</p> <pre>****</pre>		

長さ 4 のものは

```
Python3 1 print("****")
```

とすれば十分だが、70 にするには ``\*`` を 70 個正確に入力する必要があるが生じて苦痛である。そこで for の繰り返しを応用する。

```
Python3 1 for i in range(70):
2     print("*", end="")
3 print()
```

このように書くと ``70`` の部分を変更するだけで任意の長さに変更可能である。ここで `range(70)` は `range(0, 70)` の省略記法である。一般に `n` 回繰り返したい場合は `range(n)` と書くと良い。

なお for の繰り返しの意味は先ほどと変わらないが、繰り返しの内部で `i` が使われていないので、「`i` の部分を 1,2,3 と次々と置き換えながら」という部分は意味を持たず、単に 70 回繰り返す指示と等価である。また `print("*", end="")` の `end=""` は、`print` 実行終了時に行われる改行を抑制するという意味である。

例題	長方形 (rectangle)	(金子)
<p>以下のような長方形を表示せよ。以下の例では 4x3 だが 70x20 など巨大なものも簡単な変更で書けるように作成し、実際に 70x20 で表示してみよ。</p> <pre>**** **** ****</pre>		

長さ 4 を 3 行であれば

```
Python3 1 print("****")
        2 print("****")
        3 print("****")
```

と書くこともできるが、まず行数を簡単に変更可能とするために、1行ずつ for の繰り返しを応用する。

```
Python3 1 for i in range(3):
        2     print("****")
```

これは改行部分を別にして以下のようにも書くことができる。

```
Python3 1 for i in range(3):
        2     print("****")
```

さらに内側の星の表示部分を「棒」の例題のように、for の繰り返しを応用する。

```
Python3 1 for i in range(3):
        2     for j in range(4):
        3         print("*", end=" ")
        4     print()
```

このような二重の繰り返しを 二重ループ と呼ぶ。二重ループでは、内側のループがまず実行されてから外側のループがひとつ進む。

例題	三角 1 (triangle)	(金子)
<p>以下のような三角形を表示するものを定義せよ。例では大きさが3であるが、大きさ5,6なども少しの簡単に試せるように作成せよ。</p> <pre>* ** ***</pre>		

考え方: 長方形との差は、行ごとに長さが異なる点である。行の長さは1..4の4で決まっているので、この4を行に対応させれば良い

```
Python3 1 for i in range(3):
        2     for j in range(i+1):
        3         print("*", end=" ")
        4     print()
```

この例のように、range(i) の i の部分は数に対応するものを書くことができる。range(i) に代えて、range(i\*4-3) とするとどうなるだろうか。実行して説明せよ。

## 1.6 条件分岐

別の制御構造として、文を実行したりしなかったりする文法 `if` を導入する。

`if` 条件:

条件が成り立ったら実行する文

`else`:

条件が成り立たなかったら実行する文

例題	偶数奇数 (oddeven)	(金子)
<p>1 から 10 までの整数について、偶数であれば even, 奇数であれば odd と表示するプログラムを表示せよ。</p> <pre> 1 odd 2 even (中略) 10 even </pre>		

```

Python3 1 for i in range(1,11):
          print(i, end=" ")
          print("_", end=" ")
          if i % 2 == 0:
              print("even")
          else:
              print("odd")

```

(条件分岐は次週以降も取り扱うので、困難を感じた場合は次週以降にもう一度取り組んでも良い。)

## 1.7 総合問題

問題	九九 (p99)	(山口和)
<p>九九の表を表示するプログラムを作成せよ。整数 <code>i</code> と <code>j</code> の積は <code>i*j</code> で得られる。その表示は <code>print(i*j, end=" ")</code> で得られる。空白は <code>print(" ", end=" ")</code> と表示する。</p> <p>(桁をそろえたい場合は <code>print(" {0:2d}".format(i*j))</code> などのように書式指定を使うことができるが、揃えなくても良い。)</p>		

■定数 この資料では、**定数**には大文字で始まる名前を用いる。定数とは、数値の別名で、一度定義したらプログラム中からは変更できない、という概念である。実際のところ、python では変更できてしまうので、プロ

グラム作成者が注意しなければならない。多くの言語では、変更不可の定数を言語機能で実現している<sup>\*2</sup>。

Python3 1 N = 3

## 問題

## 三角 2 (triangle2)

(山口和)

以下のような三角形を表示するものを定義せよ。例では大きさが 4 であるが、大きさ 5,6 など  
も少しの変更で簡単に試せるように作成せよ。

```
*****
****
***
**
*
```

## 問題

## 矢印 (arrow)

(山口和)

以下のような矢印を描くプログラムを作成せよ。前の問題同様に、大きさを簡単に換えられる  
ように作成すること。矢印は、整数  $N$  に対して、幅  $2N + 1$  高さ  $2N + 1$  を取る。例は  $N = 3$  の  
結果である。

```
  *
 ***
*****
*****
  *
  *
  *
```

## 問題

## 森 \* (forest)

(山口和)

[3, 5, 7, 8] などの数値列を与えられて、対応する大きさの木を横に並べて描くプログラムを  
作成せよ。ただし、1 本の木は上の arrow とする。

(配列は次回扱うので、分かる範囲で作成せよ)

<sup>\*2</sup> C++ の場合は const 修飾子を用いる

## 1.8 この章の課題

提出対象: 以下のいずれか

- 問題 (★ なし) を全て Python で解く (通常コース)
- ★ つきの問題を解く (通常コースで、回答数を節約したい人)

提出先: ITC-LMS

提出時期:

- 授業終了時: (推奨) できたところまで提出する。出席の確認にも用いる
  - 完成した小問については、授業時間中にスタッフ (教員または TA) にチェックを受けて OK をもらう。スタッフ名をコメントに付記する (出席の確認にも用いる)。
  - 途中の場合は、作成途中と付記する。
- 標準提出期限: 4/13(月) なるべく完成させて提出する。でき具合をコメントする。(場合によっては再提出になる)
- 学期終了時: 最終的な成績は最後に提出されたものを用いる。

提出条件:

- 提出ファイルは、1.A 節を参考にして、[jupyter から作成した](#) weekN- (学生証番号) .ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中で完成品か、授業中に既に OK をもらっていればその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など、(途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む。(何も苦労がなかった場合を除き、自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

## 1.A 毎週の課題の提出方式について

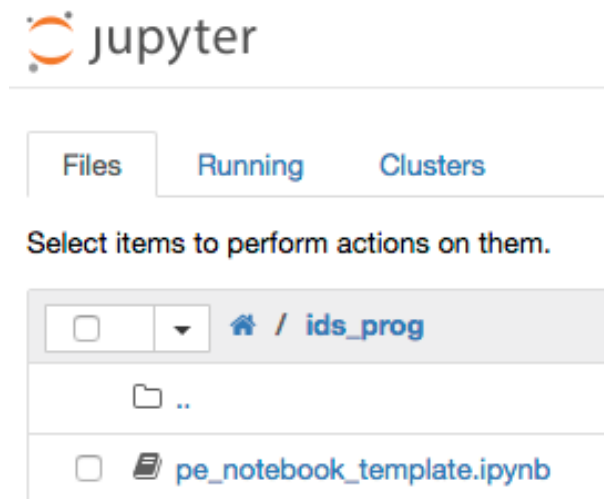
授業開始時: テンプレートの入手

ITC-LMS の本授業のページにアクセスし、「教材」に置いてある `pe.notebook.template.ipynb` をダウンロードする。提出用テンプレートを適当な場所 (例えば、ホームディレクトリ直下に `ids_prog` などの名前のディレクトリを作成する) へ移動する。

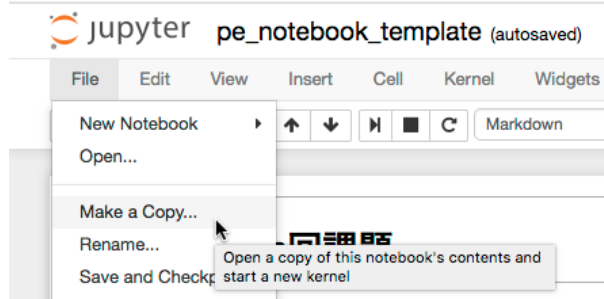
テンプレートから新しい notebook (作業用) を作る

授業中にプログラムを実行したり作業するための作業用 notebook を作成する。

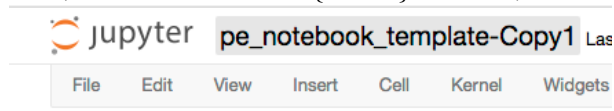
jupyter notebook 上でテンプレートファイルを開く。(jupyter notebook を起動した時に現れるブラウザ上で、適切なフォルダを選択して該当ファイルをクリックする)



File > Make a Copy でテンプレートファイルをコピーしたファイルを作成する。



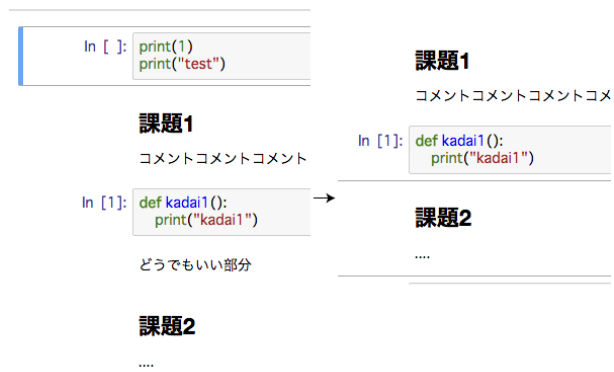
画面上の pe\_notebook\_template-Copy1 をクリックし、新しいファイル名 (weekN-work) へと変更する。(ただし N は章の番号={0,1,2,..} とする。)



### 提出時: 作業用 notebook から提出用ノートブックを作成する

先程と同様に、File > Make a Copy で weekN-work をコピーし、weekN- (学生証番号) へと名前を変更する。“ (学生証番号)” の部分を自分の番号に置き換える。自分が提出したい部分以外を削除する

「課題 1」などの見出しなどは、ダブルクリックで編集できる。文法は Markdown 記法である (必要に応じて調査せよ)。



以上の操作で作成した jupyter notebook ファイル (weekN- (学生証番号) .ipynb) を ITC-LMS に提出せよ。



## 第 2 章

# 式と評価, 関数

## *Expression, Evaluation, and Function* 2020-04-15 Wed

### 事務連絡

第 1,2 週の授業への参加に困難がある (あった) 場合は, 状況を担当教員に相談してください.  
Zoom について, なるべく, (1)profile で本名 (full name) を登録, (2)@g.ecc.u-tokyo.ac.jp のアカウントでログイン, としてから授業に参加してください.  
ITC-LMS のアンケート「Zoom 電子メールアドレスの登録」に回答してください (締切 4/20(月)).

### 概要

(前章で説明した「プログラムは文の列である」という理解の仕方とは別に) プログラムは式 (expression) を組み合わせたものであり, 式を評価する (evaluate) ことで計算が進行するという考え方を紹介する.  
なるべく授業時間中に例題や問題に多く取り組んで, 教員にチェックを受けること. 時間外等でやむを得ずチェックを受けずに提出する場合は, 関数の動作をどのように確認したかを具体的にかつ十分なだけ添えること.

## 2.1 式の評価

jupyter notebook で, 式 (expression) を入力すると 評価 (evaluate) した結果を返す:

1+1 (Shift+Enterを押す)	1
=> 2 ... pythonの返答	2

ここで 1+1 を式, 2 を値 (あたゝい,value) という. また値は型 (type) を持ち, 2 の型は整数である.

重要: 評価して値を得ることと, 表示 (print) することを区別すること. jupyter 上の見た目は似ているが, 差がある.

## 2.2 関数的な計算: 式と評価

約束

本日は、変数への値の代入 ( $\neq$  初期化) は用いないこと。また、先週説明した `for` も用いないこと。  
`print` はデバッグの表示用途のみ使って良い。

式は、原始的な式 (atomic expression) とそれを組み合わせたもの (compound expression) である。

### 2.2.1 原始的な式

数は原始的な式である。(例: 345, 3.1415)

真 (True) と偽 (False) の真偽値 (boolean) や, "Hello" などの文字列 (string) も原始的な式である。

### 2.2.2 式の組み合わせ

二項演算子 (binary operator) を使って式を組み合わせることができる。Python や C++ の多くの式は、

(式 演算子 式)

と演算子を中央に配置する (中置記法)。例:  $450 * 1.05$

このような式をさらに組み合わせたものもまた、式である。式を組み合わせでできた式について、部品となる式を部分式という。たとえば、整数と、四則演算  $+$ ,  $-$ ,  $*$ ,  $/$ , 剰余  $\%$ , 単項  $-$ ,  $()$  を自然に組み合わせたものは式である。

通常は括弧を省略する。1-2-3-4-5 は  $((((1-2)-3)-4)-5)$  という意味である。一意に括弧を復元できるように、演算子の優先順位 (e.g., 乗算は加算に優先する) や同じ順位の場合は左から評価するなどの規則が、プログラミング言語ごとに定められている。

同じ名前の演算であっても、対象となる式の持つ型 (type) 毎に異なる意味を持つ。たとえば、Python を含む多くの言語で、 $+$  演算子は、整数型については和を、文字列型については連結の効果を持つ。また、C++ などのプログラミング言語では整数の除算は整数を返すことがある (例えば、C++ では  $5/3$  と  $5/3.0$  で異なる値を返す)。Python3 では、整数と浮動小数点数の両方で除算演算子  $/$  は浮動小数点数の演算として行われる。整数の除算の商を求めたい場合は、演算子  $//$  を用いて  $5//3$  という式を書く。

### 2.2.3 変数: 値に名前をつける

変数 (variable) を使うことで計算された値 (value) に名前をつけることができる。名前に使用可能な文字は言語によって異なる。この資料では特別な場合を除いて、先頭を英字 (アルファベット), 2 文字目以降には英字の他に数字とアンダースコア ( $_$ ) を用いる。

変数を用いることで、次のような利益がある。

- 意図が不明な値に名前を付けることができるため、プログラムの意味がより分かりやすくなる。(具体的な値を使っている場合よりも、より抽象化されたプログラムにすることができる)
- 同じ計算式が何度も使われる場合に、変数によって置き換えることで式が簡単になる

- 何度も使われる値を変更する際に、変数の定義だけを変更すればよくなる

値が変更されない変数のことを特に定数 (constant) と呼ぶ。この資料の Python ではアルファベット大文字で始める。C++ では `const` 修飾子を付与する。

```
Python3 1 Size=2
```

```
C++ 1 const int Size=2;
```

と書くと、`Size` という名前の変数と `2` という値が結びつけられる。以降、`2` と書くかわりに `Size` と書いても全く同じ値が得られる。つまり、変数もまた式である。

値に名前を付けることは抽象化の第一歩である。例えば、

```
Python3 1 Pi=3.1415
        2 Radius=2
```

のような変数定義をしたとする。すると、

```
Python3 1 3.14 * 2 * 2          # (円周)
        2 3.14 * 2 * 2          # (面積)
```

と書いたのに対して

```
Python3 1 Pi * 2 * Radius      # (円周)
        2 Pi * Radius * Radius # (面積)
```

と、より意味を明瞭に書くことができる。

後者の書き方はプログラムの保守性を向上させる。たとえば半径を `2` から `10` に変更したい場合を考えよう。

## 2.2.4 関数: 計算に名前をつける

計算の手順にも名前を付けることができ、これを関数 (function) と呼ぶ。関数は、

```
def 名前 (仮引数, 仮引数...):
    return 式
```

のように定義して、

```
名前 (引数, 引数...)
```

のようにして使う。キーワード `return` の右に書いた式が、関数を評価した際の値となる。

例えば、3 次元空間での距離の自乗を計算する  $(x * x) + (y * y) + (z * z)$  という式を考える。この式には、「何かを二乗する」計算が頻出している。つまり、自乗の対象は `x` や `y` などと異なるが、自乗という手順は共通である。そこで、変化する部分を引数 `a` とおき、残る共通部分である「何かを二乗する」部分を関数としてまとめてみよう。

```
Python3
```

```

1 def square(a):
2     return a*a

```

すると、先の式は `square(x) + square(y) + square(z)` と書くことと等価である。つまり、関数も式を構成する要素として利用可能である。

関数を定義すれば、その関数の名前と「何を計算するか」さえ覚えておけば、複雑な定義の詳細を忘れてしまいうことができる。これは、計算手順の抽象化の第一歩である。

C++

```

1 int square(int x) {
2     return x*x;
3 }

```

例題

長方形の面積 (rectarea)

(金子)

二辺の長さがそれぞれ  $a, b$  である長方形の面積をもとめる関数 `rectarea(a, b)` を定義せよ

関数を作成したら、具体的な値で実行結果を確認しよう

Python3

```

1 >>> rectarea(2,3)
2 6
3 >>> rectarea(10,5)
4 50

```

例題

距離 (distance)

(増原)

座標  $(x, y)$  上の点の原点からの距離を返す関数 `distance(x, y)` を定義せよ。

平方根は `math.sqrt()` という関数が用意されている。ただし事前に `import math` という使用宣言を行う。

Python3

```

1 >>> import math
2 >>> math.sqrt(3)
3 1.7320508075688772

```

これ以降の問題も、自身で同様のテストを行って確認せよ。

### 2.2.5 関数内の変数

関数の中でも、値に名前をつける目的で変数を使うことができる。

```

def 名前(仮引数1, 仮引数2...):
    変数1 = 式
    変数2 = 式
    ...
    return 式 # 式には, 仮引数に加えて変数 1, 2... を利用可能

```

## 例題

## 三角形の面積 (triangle\_area)

(アルゴリズム入門)

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。面積は数式  $\sqrt{s(s-a)(s-b)(s-c)}$  で与えられる。ただし  $s = (a + b + c)/2$  である。

## 回答例

```

Python3 1 def triangle_area(a, b, c):
        2     s = (a+b+c)/2.0
        3     return # ここに s, a, b, c を使った面積の式を書く

```

## 2.2.6 様々な演算

■真偽値 二つの整数 (あるいは文字列) を比較演算子 (等しい`==`, 等しくない`!=`, 大きい`>`, 大きいか等しい`>=`, 小さい`<`, `<=`) で結んだ式を評価すると 真偽値 (Boolean) が得られる。

```

>>> 3 < 5
True
>>> 3+5 < 7
False
>>>

```

また真偽値同士の演算として, 論理積 `and`, 論理和 `or`, 単項演算として否定 `not` が用意されている。なお C++ などではそれぞれ `&&`, `||`, `!` の記号を用いる。

## 例題

## 除算 (is\_divisible)

(アルゴリズム入門)

整数  $x$  が  $y$  で割り切れる場合真, そうでなければ偽を返す関数 `is_divisible(x, y)` を定義せよ。

例題	Range (in_range)	(AOJ Introduction to Programming I)
3つの整数 $a, b, c$ に対して, それらが $a < b < c$ の条件を満たすならば True を, 満たさないならば False を返す関数 <code>in_range(a, b, c)</code> を作成せよ		

ヒント: Python では, 比較演算子を chain することが出来る (!)\*<sup>1</sup>. すなわち,  $a < b < c$  と書くと期待通りの真偽値を得られる.

一方, 他の言語 Ruby や C++ では, 二項演算子では3つの数を同時に比較することができない. その場合は,  $a < b$  と  $b < c$  をそれぞれ判定してから, 論理積を取る.

例題	ほぼ一致 (close_enough)	(増原)
2つの数 $(x, y)$ が十分に近い場合に真, そうでない場合に偽を返す関数 <code>close_enough(x, y)</code> (十分に近いとは, 定数 $\Delta > 0$ を使って $ \frac{x-y}{y}  < \Delta$ である場合とする. $ y  > \Delta$ は仮定して良い. $\Delta$ は自分で適当な値に定める)		

例題	平方根判定 (is_sqrt)	(増原)
数 $x, y$ を受け取り, $x$ が $y$ の平方根になっていれば真, そうでなければ偽を返す <code>is_sqrt(x, y)</code> (ただし, 数値誤差を考慮して, $x^2$ と $y$ が十分に近いことを判定することにする. 直前の例題参照)		

例題	倍数 (multiple_of_two_or_three)	(アルゴリズム入門)
与えられた数が2または3の倍数であるときに真, そうでないときに偽となる関数 <code>multiple_of_two_or_three(x)</code>		

問題	Circle in a Rectangle (in_rectangle)	(AOJ Introduction to Programming I)
<p>長方形の中に円が含まれるかを判定する関数 <code>in_rectangle(w, h, x, y, r)</code> を作成せよ. 長方形は左下の頂点を原点とし, 右上の頂点の座標が <math>(w, h)</math> である. 円は中心の座標が <math>(x, y)</math> 半径が <math>r</math> である. 円が含まれるなら True を, そうでなければ False を返すとする.</p> <p>図解 <a href="http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_2_D">http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_2_D</a></p>		

\*1 <https://docs.python.jp/3/reference/expressions.html#comparisons>

■文字列 "Hello"のように二重引用符で囲まれたものは文字列である。文字列に対しては、+ 演算子が結合の意味を、\*演算子が+ 演算子を与えられた数だけ繰り返す意味を持つ。

```
>>> "Hello" + "World" 1
'HelloWorld' 2
>>> "Hello"*3 3
'HelloHelloHello' 4
```

```
>>> "Hello".upper() 1
'HELLO' 2
```

ドットはメソッド呼び出しを表す。この演習内では upper("Hello") のように関数の特殊表記として理解してよい。

問題	Watch (watch)	(AOJ Introduction to Programming I)
<p>0 時からの経過秒数 <math>s</math> に対応する hh:mm:ss 形式の文字列を返す関数 watch(<math>s</math>) を定義せよ。数値が1桁の場合、0 を付けて 2桁表示をする必要はない。(後の方を参考にすれば0をつけてもよい)</p> <p>watch(46979) は 13:2:59 という文字列を返す。</p>		

実行例補足: watch(46979)+"!?" を評価すると"13:2:59!?" となる。

整数の 1 と 2 をコロンでつなげた文字列は str(1)+" ":"+str(2) で与えられる。整数は str という関数により文字列に変換される。

```
>>> 1 1
1 2
>>> str(1) 3
'1' 4
>>> str(3+5) 5
'8' 6
```

C++ の場合は、std::to\_string という標準関数が、整数を文字列(std::string)に変換するために用意されている。ただし、C++11 という比較的新しい仕様で加わったものなので、コンパイルオプションに -std=c++11 を追加する。(環境によってはなくても動く場合もあるかもしれない。)

```
C++11
1 #include <string>
2 #include <iostream>
3 using namespace std;
4 // 整数 a と b をコロンでつなげた文字列を返す関数
```

```

5  string concat(int a, int b) {
6      return to_string(a) + ":" + to_string(b);
7  }
8  // 実行例
9  int main() {
10     cout << concat(3,5) << endl;
11 }

```

問題	Abbreviation (abbreviation)	(増原)
与えられた文字列の (1) 先頭 1 文字, (2) 「(文字列全体の長さ) - 2」という値の整数 (を文字列に変換したもの), (3) 最後の 1 文字をつなげたものを返す関数 <code>abbreviate(word)</code> を定義せよ。例えば, "internationalization" であれば "i18n" を返す。		

文字列  $x$  に対して, 先頭の文字は  $x[0]$ , 末尾の文字は  $x[-1]$  で得られる。また,  $\text{len}(x)$  は文字列の長さを与える。

C++ では, `string` 型の文字列  $x$  に対して,  $x.\text{substr}(n, 1)$  が  $x$  の  $n$  文字目 (からはじまる 1 文字の) 文字列を,  $x.\text{size}()$  が  $x$  の文字数を返す。

実行例補足: `abbreviate("internationalization")+"!!!"` を評価すると `"i18n!!!"` となる。

## 2.2.7 条件判断

条件によって計算の内容を変えることによって, より複雑な計算をさせることができる。if と else は条件によって式を切り替える文法である。

<b>if</b> (条件: 真偽値になる式):	1
<b>return</b> 条件が真の場合の式	2
<b>else:</b>	3
<b>return</b> 条件が偽の場合の式	4

例えば, 絶対値を求める関数は数学的には

$$|x| = \begin{cases} x & (x \geq 0) \\ -x & (\text{otherwise}) \end{cases} \quad (2.1)$$

であるが, これが次のように書ける:

```

Python3 1 def abs(x):
2         if x >= 0:
3             return x
4         else:
5             return -x

```

条件が成り立たない場合に更に `elif` を用いて他の条件で場合分けすることもできる:



```

if (条件1):
    return 条件1が真の場合の式
elif (条件2):
    return 条件2が真の場合の式
elif (条件3):
    return 条件3が真の場合の式
...
else:
    return どの条件も偽の場合の式

```

C++

```

1 int abs(int x) {
2     if (x >= 0)
3         return x
4     else
5         return -x
6 }

```

例題

点数を成績に (grade)

(山口和)

関数 `grade(g)` として,  $g$  が 80 以上なら "A",  $g$  が 65 以上なら "B",  $g$  が 50 以上なら "C", それ以外なら "D" を返す関数を定義せよ.

例題

AMPM (meridiem\_hour)

(増原)

与えられた時刻が午前 (0~11) だった場合 "am" という文字列を, 午後 (12~23) だった場合 "pm" という文字列を返す関数 `meridiem_hour(h)` を定義せよ.  
(それ以外の値には "error" という文字列を返すとする.)

例題

 $3n+1$  (collatz\_step)

(アルゴリズム入門)

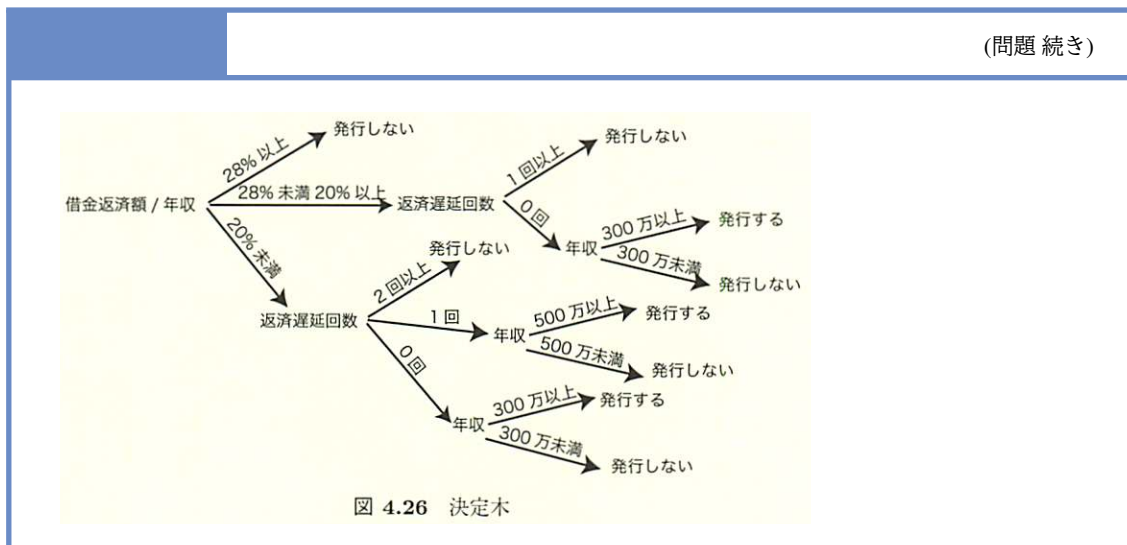
$n$  が偶数の場合に  $n/2$  を,  $n$  が奇数の場合  $3n+1$  を返す関数 `collatz_step(n)` を定義せよ.

問題

信用 (issueCC)

(山口和)

(年収に対する) 借金返済額  $dt$ , 返済遅延回数  $dl$ , 年収の額  $ic$  に対して, 以下の決定木に従って, クレジットカードを発行するかどうかを真偽値 (True か False) で返す関数 `issueCC(dt, dl, ic)` を定義せよ.



注: True は, 文字列 "True" とは異なる.

## 問題

## Simple Calculator (calculator)

(AOJ Introduction to Programming I)

二つの整数  $a, b$  と四則演算  $op$  に対応する値を返す関数  $calculator(a, op, b)$  を作成せよ.  
たとえば  $calculator(1, "+", 2)$  は 3 を返し,  $calculator(3, "/", 2)$  は 1 を返す.

注記:  $calculator(calculator(3, "*", 10), "+", 7)$  は,  $3 \cdot 10 + 7$  を計算し 37 を返す.

## 2.2.8 計算の順序

複雑な式を評価するとき, 実際にはどのようにして計算が行われているのだろうか?

例えば「 $(3 * 3) + (4 * 4)$ 」のような式からどうやって 25 が求められたり,  $square(8)$  という式からどうやって 64 が求められているのだろうか.

実際には, プログラミング言語毎に様々な規則があり, 簡単には説明できない. しかし, 本章で扱ったような式を中心とした関数的な (副作用を含まない) プログラムの場合は, 以下のように, 比較的簡単なモデルで演算した場合と一致するように作られている.

■原始的な演算子の場合: 式を評価する手順は,

- まず部分式を全て評価する.
- 演算子に部分式を評価した値を適用 (apply) する.  
「適用する」とは, + だったら実際に加算を行うことであり, 計算機の内部では「加算をする機械命令」が実行される.

のように定義できる. この定義は (後で述べるように) 再帰的なものである.

■関数を含む場合: 関数を部分式として含む式の場合, 部分式全体を関数の定義部分で置き換えて評価を進める。その際に, 仮引数の変数は実引数によって置き換える。

例えば,

```
Python3 1 def double(x):
        2     return x+x
```

という定義があったときに, `double(3)` という式を評価すると, 「`double(3)`」を,  $x = 3$  に注意して,  $3 + 3$  という式に置き換えて評価が続けられる。

こみいった式の場合は, どの式から置き換えをしてゆくか選択肢がある。Python や C++ では内側の式, 関数の内側の引数や複式の評価が行われてから外側の式が評価される。

```
square(double(3))
```

とした場合に

```
square(double(3))
square(3+3)
square(6)
6*6
```

のように計算が進む。

■短絡評価: 効率的な評価のために, 論理演算では「左から評価を始めて, 答えが決まったらそこで打ち切る」というルールが多くの言語で導入されている。たとえば (例のための例で現実感がないが)

```
x = (square(5) == 30) and (square(6) == 36)
```

と書いた場合, `square(5)` の計算は行われるが, `square(6)` の計算は行われない。これは論理積演算子の左側の項 (`square(5) == 30`) が偽となったため, 右の項の真偽値によらず式全体の値が偽となることが分かったためである。これにより, 無駄な計算を省くことができる。

比較として

```
x = square(5) + square(6)
```

という演算では, `square(5)` と `square(6)` がまず計算されてから, `+` の計算が行われる。

## 2.3 参考: 命令的な計算

前の section で見てきた計算は, 四則演算のように何度計算しても結果が変わらないようなものであった。このようなものを関数的 (functional) な計算という。

現実世界のプログラムでは, 計算をする度に結果が変わるものや, 計算をした結果, 外界に影響を与えるものごとも扱う必要がある。このようなものを命令的 (imperative) な計算と言う。

例:

- キーボードから 1 行分の文字列を入力をする: `input()`
- 画面へ文字列を出力する: `print("To be or not to be ...")`
- 乱数の発生: `random.random()`

このような計算を扱う場合には、実行の順番に依存して結果が変わる。そのために命令的な実行を扱うプログラミング言語では、前章で紹介した「文」という実行順序に関する概念を設けて、「原則として前から順に実行される」というルールを導入する。今後この資料では、文とは、`print` だけでなく、すべての式、変数の初期化、代入を含む。

関数的に記述したプログラムのほうが、動作確認の手間が圧倒的に少ない。そこで命令的な計算の使用は小規模に留めて、なるべく関数的な計算でプログラムを構成することがお勧めである。

命令的なプログラムを扱う場合は、プログラミング言語の仕様に対する正確な理解が必要となる場面が多い:

```
C++
1  int a() {
2      cout << "a" << endl; // 1 を返すついでに a と表示
3      return 1;
4  }
5  int b() {
6      cout << "b" << endl; // 1 を返すついでに b と表示
7      return 1;
8  }
9  int sum(int x, int y) {
10     return x + y;
11 }
12 int main() {
13     int v0 = a() + b();
14     int v1 = sum(a(), b());
15     cout << v0 << endl;
16 }
```

たとえば部分式の評価順序は C++ では規定されていないので、`a() + b()` であっても `sum(a(), b())`; であっても、`a()` と `b()` がどちらが先に評価されるかは環境依存である。式の値は評価順序によらず `1+1=2` であるが、`a` と `b` の表示順序は異なりうる。

```
C++
1  int p = a();
2  int q = b();
3  cout << p+q << endl;
```

予期しない事態を避けるためには、1 文に含まれる式を単純に保つことは有効である。

## 2.4 (発展) 関数を返す関数

比較的新しいプログラミング言語では関数自身を関数の引数に渡したり、関数の返り値とする機構が用意されている。

説明の簡潔さのため、単独の引数 `x` をとる関数を以下では考える。

Python3

---

```
1 def xplus3(x):
2     return x + 3
```

---

```
>>> xplus3(1) 1
4 2
```

---

同様の関数を、`lambda` を用いて実行時に作成することもできる。これを匿名関数と呼ぶことにする。関数の呼び出しの際には (通常関数と同様に呼び出せる。

```
>>> f = lambda x: x+3 1
>>> f(1) # 1+3 2
4 3
>>> f(2) # 2+3 4
5 5
```

---

以下のように複数の引数を取ることもできる。

---

```
lambda 引数1, 引数2, ... : 単一の式 1
```

---

`lambda` で作成した関数を、関数の返り値にすることができる。次の関数 `plusn` は、『「引数 `x`」に `n` を足して返すような匿名関数』を作成して返す。

**Python3**

---

```
1 def plusn(n):
2     return lambda x: x+n
```

---

```
>>> f = plusn(10) 1
>>> f(1) 2
11 3
>>> g = plusn(-10) 4
>>> g(1) 5
-9 6
```

---

`lambda` では、単一の文しか書くことができない。複数行の文を書きたい場合は、`def` による関数定義を使う。

**Python3**

---

```
1 def xplus3():
2     def inner_plus3(x):
3         y = x+3
4         return y
5     return inner_plus3
```

---

匿名関数を, 別の関数に渡して組み合わせた関数を作成することもできる.

Python3

```
1 def compose_product(f, g):          #  $h(x) = f(x) * g(x)$ 
2     return lambda x: f(x) * g(x)
```

```
>>> h = compose_product(f, g)      1
>>> h(1)                            2
-99                                3
>>> f(1)                            4
11                                5
>>> g(1)                            6
-9                                7
```

問題

変数を含む電卓 \* (calcvar)

(金子)

変数  $x$  を含む四則演算の式を表す文字列  $s$  を受け取って, 引数  $x$  に応じた値を計算する関数を返す関数 `parse(s)` を作成せよ.

簡単のため文字列  $s$  は以下の条件を満たす

- 演算は, Simple Calculator で扱う四則演算のみである
- 数字は, 0 から 9 までの 1 文字の整数のみが登場する.
- 変数は, 現れない場合もある. 現れる場合も  $x$  のみである.
- 演算子一つにつきカッコが一組つく. 余計なカッコはつかない

この課題は, 全員の義務ではない. 以下にヒントも用意されているが, 再帰や多重代入の利用が想定されるので, 経験がない場合は第 5 週以降に考えるのが適しているかもしれない. この課題に限っては C++ を用いても良い. ただし Python の `eval` は使用禁止とする.

```
>>> f = parse("((x+3)+3)")          1
>>> f(0)                            2
6                                    3
>>> f(1)                            4
7                                    5
>>> f = parse("((x+3)*(x-3))")      6
>>> f(1)                            7
=> -8                               8
```

部品ごとに分けて作ると良い. 入力のような式も, 引数  $x$  を受けて値を返すような匿名関数の組み合わせで, 対応できることに注意する.

1. 引数  $c$  が  $x$  一文字か数字位置文字だと仮定する.  $c$  に対応する関数を返す関数 `make_term(c)` を作成せよ. 文字が  $x$  の場合は identity function を, 数値の場合は定数関数を返すとする.

```

>>> f = make_term("x") # f(x) = x      1
>>> f(3)                      2
3                                3
>>> f(5)                      4
5                                5
>>> f = make_term("9") # f(x) = 9      6
>>> f(3)                      7
9                                8

```

2. 一つの演算子を含む長さ3の文字列を引数として、対応する関数を返す `parse_one(s)` を作成せよ。以下のような構造になるはずである。また問題 Simple Calculator で作成した関数を呼び出すと簡潔になる。

```

Python3 1 def parse_one(s):                # "x+3" => xplus3
2     lf = ... # s[0] に対応する関数
3     op = s[1] # operator
4     rf = ... # s[2] に対応する関数
5     def func(x):
6         ... # lf, rf, op と x から値を計算する, 組み合わせた関数を定義
7     return func

```

```

>>> f = parse_one("x+3")      1
>>> f(100)                    2
103                           3

```

3. (`parse_one` を複製改変して)1重のカッコを含む式(例 `"(x+3)"`)に対応する匿名関数を返す関数 `parse_term(s)` を作成せよ。戻り値は長さ2の配列とし、先頭要素が対応する関数、次の要素が閉じカッコの位置とする。

```

>>> s = "(x+3)blahblahblah"      1
>>> f, pos = parse_term(s) # 結果を, f と pos の二つの変数で受け取る 2
>>> f(3)                          3
6                                  4
>>> pos                          5
5                                  6
>>> s[pos:] # 部分文字列, "(x+3)"を解析した残りに相当 7
'blahblahblah'                    8

```

4. `parse_term(s)` を変更し、ネストするカッコを含む式に対応させよ。たとえば以下のような再帰で作ることができる。

```

Python3 1 def parse_term(s):
2     if s[0] == "(":                # カッコを含む式

```

```

3         # 再帰で、開きカッコの右から演算子の左側までを解析
4         lf, lsize = parse_term(s[1:])
5         op = s[lsize+1] # 演算子
6         rf, rsize = parse_term(s[...]) # 再帰で、演算子の右側を解析
7         f = lambda x: ...
8         return f, ...
9     else:                                     # 1文字の式
10         f = make_term(s[0])
11         return f, 1

```

5. `parse_term(s)` を利用して、問題に対応する関数 `parse(s)` を定義する

```

Python3 1 def parse(s):
2         f, size = parse_term(s)
3         return f

```

C++ の場合も新しい規格である C++11 の機能を用いると、以下のように匿名関数を扱うことができる (コンパイルオプションに `-std=c++11` をつける). `double` を引数にとり `double` を返す関数の型は, `function<double(double)>` である. 匿名関数の作成は, `[] (引数) { 本体 }` という文法に従う. 冒頭の `[]` 内では, 本体の式で扱う外側の変数を列挙する. 詳しくは文法書を参照のこと. `auto` は型推論を利用した型の略記法である. 関数呼び出しの際は, 引数をカッコで直接渡す.

```

C++11 1 #include <functional>
2 #include <iostream>
3 using namespace std;
4 function<double(double)> compose_product(function<double(double)> f,
5         function<double(double)> g)
6 {
7     return [f,g](double x){ return f(x)*g(x); };
8 }
9 int main() {
10     auto f = [](double x){ return x+3; };
11     auto g = [](double x){ return 2*x; };
12     auto h = compose_product(f, g);
13     cout << h(3) << endl;    // 6*6 = 36
14     cout << h(5) << endl;    // 8*10 = 80
15 }

```

## 2.5 この章の提出課題

提出対象: 以下のいずれか

- 問題(★なし)を全て Python で解く (通常コース)
- ★つきの問題を解く (通常コースに苦勞がない場合)

提出先: ITC-LMS



提出時期:

- 授業終了時: (推奨) できたところまで提出する。出席の確認にも用いる
  - 完成した小問については、授業時間中にスタッフ (教員または TA) にチェックを受けて OK をもらう。スタッフ名をコメントに付記する (出席の確認にも用いる)。
  - 途中の場合は、作成途中と付記する。
- 標準提出期限: 4/20(月) なるべく完成させて提出する。でき具合をコメントする。(場合によっては再提出になる)
- 学期終了時: 最終的な成績は最後に提出されたものを用いる。

提出条件:

- 指定した課題が、指定した関数名で作成されていること
- 提出ファイルは、1.A 節を参考にして、[jupyter から作成した](#) weekN- (学生証番号) .ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中で完成品か、授業中に既に OK をもらっていればその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など、(途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む。(何も苦労がなかった場合を除き、自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

## 2.A ターミナルでの Python3 の利用

ターミナルで python を使う場合は、python3 に切り替えて使う (python2 は忘れること。python2 での提出は認めない)。下記を一度行くと、次回から自動で python3 が利用可能になる。

```
host:~ 12345$ pyenv local anaconda3-4.4.0 1
host:~ 12345$ python3 --version 2
Python 3.6.1 :: Anaconda 4.4.0 (x86_64) 3
```

インタプリタは python3 として起動する。

```
$ python3 1
>>> 1+1 2
2 3
```

本資料では、基本的に jupyter 上での実行を勧めるが、こちらのインタプリタが役に立つこともある。

### 2.A.1 整数除算

python2 と 3 で挙動が異なるので注意。

---

\$ python2	1
>>> 3 / 5	2
0	3
>>> 3 / 5.0	4
0.6	5

---

---

\$ <b>python3</b>	1
>>> 3 / 5	2
0.6	3
>>> 3 // 5	4
0	5

---

## 2.B 参考資料

プログラミングのパラダイムに関しては, 計算機プログラムの構造と解釈 (<http://sicp.iijlab.net/fulltext/>), Concepts, Techniques, and Models of Computer Programming (<http://mitpress.mit.edu/books/concepts-techniques-and-models-computer-programming>)などを参照. 今回の資料には 1997 年と 1999 年に増原先生によって行われた演習資料から拝借して金子が改変した部分を含む.

## 第3章

# ユニットテスト，配列，ループ不変条件

*Unit test, Array and Loop invariant*  
2020-04-22 Wed

### 概要

前週は「関数的に」(functional) プログラムを書くことをおすすめした。今週は、「命令的」(imperative) な文法を再び導入する。これによりプログラムを書く自由度が向上するが、一方で、プログラムが複雑になり理解が困難となるリスクもある。そこで、プログラムを分かりやすく保つための方法を合わせて身に付けよう。最初に紹介する「ユニットテスト **ユニットテスト**」(unit test) は、プログラムを意図通り作成できたかを自動で検査する方法を提供する。続いて、配列と変数 (値の変更を許す) を題材にループ不変量 (loop invariants) という概念を学ぶ。

## 3.1 テストの自動化: ユニットテスト

先週は、関数を作成した後、jupyter 上でテストすることを行った。

(再掲)

例題

長方形の面積

(金子)

二辺の長さがそれぞれ a,b である長方形の面積をもとめる関数 `rectarea(a,b)` を定義せよ

関数をデバッグしたり、あるいは完成したあとにも高速化など変更を加える際など、同じテストを繰り返し **確実に** 行いたい。そこで、テストの作業を自動化しよう。

Python の場合は `unittest` というモジュールが標準で提供されている。なお、`if __name__ == '__main__':` はこのファイルが指定されて実行された場合に、(`import` 等でロードされた場合ではない)、のみ実行する意図の書き方。

Python3

```
1 def rectarea(a,b):
2     return a*b
3
4 import unittest # 定型句
```

```

5 class TestArea(unittest.TestCase): # 定型句
6     def test_area(self): # 名前を "test_" で始める
7         self.assertEqual(rectarea(7,5), 35)
8         self.assertEqual(rectarea(10,20), 200)
9
10 if __name__ == '__main__':
11     unittest.main(argv=['first-arg-is-ignored'], exit=False) #
    jupyter で unittest を行う場合の定型句

```

クラスの説明は後日に譲るので定型句が多いが、8-9 行目の `assertEqual` が本質である。それぞれの行は、具体的な値で計算結果を検証する、テストケースである。

実行は `jupyter` 上で行う。

```

.. 1
----- 2
Ran 1 tests in 0.003s 3
4
OK 5

```

実行結果中、`tests` が実行したメソッド(この場合は `test_area`)の数である。もし、この値が0の場合にはテストが実行できていないので、細部を確認する。

もしテストに失敗すると、以下のように出力される。

```

F 1
===== 2
FAIL: test_area (__main__.TestArea) 3
----- 4
Traceback (most recent call last): 5
  File "<ipython-input-2-0b366ee99b93>", line 9, in test_area 6
    self.assertEqual(rectarea(10,20), 201) 7
AssertionError: 200 != 201 8
9
----- 10
Ran 1 test in 0.002s 11
12
FAILED (failures=1) 13

```

テストが失敗すると、`failures` または `errors` が表示され、失敗した個数が表示される。`failures` や `errors` の表示が出ないことが期待される。もし、表示が出る場合は、関数の実装と、テストケースの両方を確認する。

テスト駆動開発という開発スタイルでは、失敗するテストを積極的に利用して開発を勧める(付録 3.A を参照)。

## 3.1.1 テストケースの作成

何をテストすれば良いかは関数の複雑さに依存する。関数 `rectarea` を乗算だけで作成していれば、1, 2 のケースでほぼ十分であろう。(ただしテストケースを欺く、「仕様を満たしていないのに、テストケースでは検出できない実装」が存在することは認識しておく必要がある。)

```
Python3 1 def area_fake(a,b): # こんなコードは来ないことを前提としている
2     if a == 5:
3         return 35
4     else:
5         return 200
```

さっそくテストケースを自分で作ってみよう。

## 例題

## issueCC-テストケース実行

(金子)

以下の(先週扱った)問題について `issueCC(30000000,0,100000000)` は、(借金返済額/年収が30%以上であるから) `False` を返すことが期待される。

```
Python3 1 self.assertFalse(issueCC(30000000,0,100000000))
```

が実行されるテストを書いて、テストをパスすることを確認せよ。もし失敗した場合は、関数 `issueCC` の定義を修正し、再度テストを実行し、テストをパスすることを確認せよ。

## 問題

## 信用(再掲)

(山口和)

(年収に対する) 借金返済額 `dt`, 返済遅延回数 `dl`, 年収の額 `ic` に対して、以下の決定木に従って、クレジットカードを発行するかどうかを `true` か `false` で返す関数 `issueCC(dt, dl, ic)` を定義せよ。

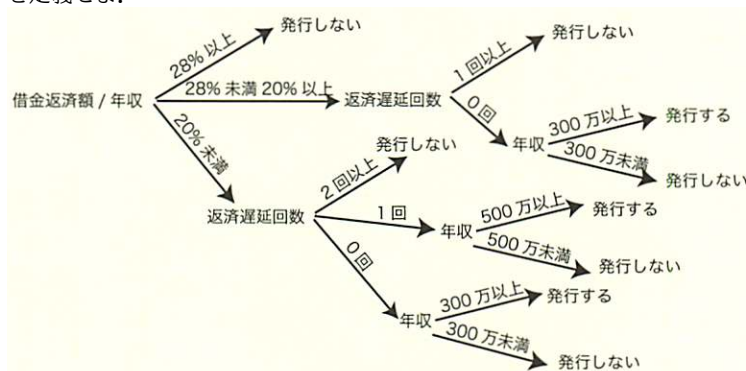


図 4.26 決定木

例題	Range-テストケース作成 (test_range)	(金子)
<p>以下の例題と回答例に対して, (1) もし正しく実装していれば成功するが, 回答例のプログラムは失敗する, というテストケースを作成せよ. (2) 実際にテストを実行し, 失敗することを確認せよ. (3) 関数 range の実装例を修正し, テストが成功することを確認せよ.</p>		
例題	Range (再掲)	(AOJ Introduction to Programming I)
<p>3つの整数 <math>a, b, c</math> に対して, それらが <math>a &lt; b &lt; c</math> の条件を満たすならば True を, 満たさないならば False を返す関数 <code>in_range(a, b, c)</code> を作成せよ</p>		
<p>回答例:</p>		
Python3	<pre>1 def in_range(a, b, c): 2     return a &lt; b or b &lt; c # 間違い</pre>	

関数 `range` については,  $a, b, c$  の大小関係や等号の有無に関して, 複数のケースを確認の方がより安心である.

問題	Circle in a Rectangle-テストケース作成 (test_in_rectangle)	(金子)
<p>先週作成した Circle in a Rectangle に対する適切なテストケースを作成せよ.</p>		

ヒント: この場合は, (i) , (ii) , (iii)-(vi) , の6通りをテストすること (一部は白文字で書いてある).

今週以降の課題提出では, ほとんどの問題に対して, テストケースを添えることが条件となる. 関数と同時にテストケースを作っておくこと.

## 3.2 配列と関数

### 3.2.1 関数内の変数 (前半は再掲)

関数の中でも, 値に名前をつける目的で変数を使うことができる.

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

---

```
return 式 # 仮引数に加えて変数 1, 2.. を利用可能
```

---

5

このように、変数は式に別名を与えるもので、その値は関数内で一意に定まるという使い方を基本とする。

### 例題

### 三角形の面積 (triangle\_area)

(アルゴリズム入門)

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。面積は数式  $\sqrt{s(s-a)(s-b)(s-c)}$  で与えられる。ただし  $s = (a + b + c)/2$  である。

### 回答例

Python3

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

---

計算の様子を知るために `print` を用いることもできる。先に書いた文から実行される。

```
def 名前(仮引数1, 仮引数2...):
    変数1 = 式1
    print(変数1)
    変数2 = 式2
    print(変数2)
    ...
    return 式 # 仮引数に加えて変数 1, 2.. を利用可能
```

## 3.2.2 配列

配列は、0 個以上の式をカンマでつなげ “[” と “]” でくくって構築したものである (例: `[3, 4, 5]`)。空の配列も存在する (例: `[]`)。

変数  $a$  が配列だとして、 $a[i]$  は  $a$  の  $i$  番目の要素を与える演算である ( $i$  は整数とする)。また `len(a)` は要素の数を与える。

```
>>> a = [3, 15+53, 10000]
>>> a
[3, 68, 10000]
>>> a[0]
3
>>> a[1]
68
>>> a[2]
10000
>>> a[3] # a の要素数は 3 なので、各要素に対応する添字は 0, 1, 2
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> a[-1] # 負の数は逆順に列を参照
10000
>>> len([10,20,30])
3
>>>

```

## 例題

## Range 配列版

(AOJ Introduction to Programming I を改題)

3つの整数を要素として持つ配列  $a$  に対して、それらが  $a_0 < a_1 < a_2$  の条件を満たすならば True を、満たさないならば False を返す関数 `in_range_array(a)` を作成せよ。なお、今回のように対応関係が自明の場合は、 $a[i]$  を  $a_i$  と断りなく表記する。

Python3

```

1 def in_range_array(a):
2     return # ここを埋める

```

```

>>> in_range_array([3,4,5])
True
>>> in_range_array([3,-4,5])
False

```

## 3.3 変数の値の書き換えとループ不変条件

既に定義済の変数の値を書き換える操作を **代入** という。

```

変数 = 式

```

既に紹介している初期化との違いは、既に定義した変数の値を書き換える点である。代入前の値にはアクセスできなくなることから、破壊的代入と呼ばれることもある。初期化と(破壊的)代入は、呼び方だけではなく実質的な区別がある。多くの言語において、定数は、初期化は可能でも代入は許されない(文法エラーとなる)。C++ では初期化にはコンストラクタ (constructor) が、代入では `operator=` がそれぞれ呼ばれ、ユーザ定義クラスでは動作を変えることもできる。

```

>>> s = 3 # 初期化
3
>>> s
3
3 # (これまで勧めていた使い方)

```



```
>>> s = 5 # 代入: この行を境に s の値は変わる      5
5                                                    6
>>> s                                              7
5 # さっきと違う                                  8
```

以下は、初心者をししばしば混乱させる、しかし文法に則った文である。

```
Python3 1 a = a + 1
```

代入では右辺を先に評価し、その値を左辺に代入する。つまり

```
Python3 1 a = 2
        2 a = a + 1
```

であれば、 $a = 2+1$  すなわち、 $a = 3$  として実行される。なお、このようなケースはしばしばあるので、次のような略記法が用意されている。

```
Python3 1 a += 1 # a = a+1 と同じ
```

代入は、理解に右辺値や左辺値など様々な概念を必要とする、難しい概念である。

本演習では、機会を限定して代入を用いることを勧める。その一つが、次に紹介するループと共に用いる場合である。また、この資料では当面、代入の左辺には変数単体、または  $a[i]$  のような配列の要素のみを扱うとする。

```
def 名前(仮引数1, 仮引数2...):      1
    変数 = 初期値                  2
    for ...                        3
        変数 = 式 # 変数を新しい値に書き換え 4
    return 変数                    5
```

#### 例題

#### 連続する数の和 (sum\_to\_n)

(山口和)

整数 1 から  $n$  までの合計を、愚直に足して求める関数 `sum_to_n(n)` を作成せよ。ただし  $1 < n$  とする。(n=10000 など検算する)

回答例:

```
Python3 1 def sum_to_n(n):
        2     sum = 0
        3     for i in range(1, n+1):
        4         sum = sum + i
        5     return sum
```

このようなプログラムがどのように導かれるか追ってみよう。

仮に固定値である 3 までの合計であれば、シンプルに記述可能である。

```
Python3 1 def sum3():
2     return 1+2+3
```

補助変数  $s_i$  (0 から  $i$  までの和) を用いて、等価なプログラムに書き直すこともできる。

```
Python3 1 def sum3():
2     s0 = 0 # 空集合の和
3     s1 = s0 + 1 # 1 の和
4     s2 = s1 + 2 # 1 と 2 の和
5     s3 = s2 + 3 # 1 と 2 の 3 の和
6     return s3
```

代入を用いると、補助変数  $s_i$  を一つの変数 `sum` で表現することもできる。

```
Python3 1 def sum3():
2     sum = 0 # sum は s0 相当
3     sum = sum + 1 # 左辺の sum は s1, 右辺の sum は s0 相当
4     sum = sum + 2
5     sum = sum + 3
6     return sum # sum は s3 相当
```

これを `for` で書き換えたものが、冒頭の回答例である。

```
Python3 1 def sum_to_n(n):
2     sum = 0
3     for i in range(1, n+1):
4         # (a) この時点で sum の値は 0 から i-1 までの和 = i(i-1)/2
5         sum = sum + i # なお, sum += i と書いても同じ
6         # (b) この時点で sum の値は 0 から i までの和 = i(i+1)/2
7     return sum
```

コメント (a)(b) に記述した `sum` の値に関する条件が、ループのどの時点でも成り立つことを確認してほしい。このような条件を、ループ不変条件という。これらの `sum` の値に関する条件を用いて、関数が  $n$  までの和を計算することを証明することができる。

次の 3.4 節の課題では、提出する全てのソースコードにループ不変条件を明確にしてコメントで記述すること。なお、説明のために上記の例では (a)(b) 二つの条件を書いたが、以下の演習の回答では、各ループ開始直後 (a) または終了直前 (b) のどちらかを記述すれば良い。



#### Loop 不偏条件の表現

動作や計算手順を表す言葉は避ける。静止した状態を表現すること。(1 を足して.. $i$  まで足したもの v.s. 1 から  $i$  までの和)

## 3.4 配列とループ

例題	要素の和 ( <b>mysum</b> )	(山口和)
配列 <code>a</code> に格納されているデータの合計を求める関数 <code>mysum(a)</code> を作成せよ。		

例題	最大値 ( <b>mymax</b> )	(山口和)
配列 <code>a</code> に格納されているデータの最大値を求める関数 <code>mymax(a)</code> を作成せよ。		

なお実際には, Python の配列には, 最大値を返す `max` という関数が用意されている。(が, この例題ではそのことは忘れるとする)

例題	最大値の添字 ( <b>max_index</b> )	(山口和)
配列 <code>a</code> に格納されているデータの最大値の添字を求める関数 <code>max_index(a)</code> を作成せよ。		

例題	二番目に大きな値 ( <b>second_max</b> )	(山口和)
配列 <code>a</code> に格納されているデータのなかで 2 番目に大きな値を求める関数 <code>second_max(a)</code> を作成せよ。		

問題	最小と最大を除いた合計 ( <b>score</b> )	(山口和)
審判の点数が配列 <code>a</code> で与えられる。それをオリンピック方式 (最大と最小の点数をつけた審判をそれぞれ一人ずつ除いた残りを使う) で合計する関数 <code>score(a)</code> を作成せよ。審判は 3 人以上と仮定して良い。		

問題	各桁の和 ( <b>digit_sum</b> )	(金子)
与えられた正の整数 <code>n</code> に対して, 十進表現での各桁の和を求める関数 <code>digit_sum(n)</code> を作成せよ。		

整数 `n` の桁数は `math.floor(math.log10(n))+1` で得ることができる (`import math` が必要)。ま

た 10 の  $n$  乗は  $10^{**n}$  で得られる。

### 3.5 配列や文字列の変更

合計を表す変数に数を足す変更が役に立ったように、配列に要素を追加したい場合もある。Python では `append` というメソッドが用意されている。

```
>>> a
[1, 2]
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4] # aの指す配列の内容は変更されている
>>> a = a + [5]
>>> a
[1, 2, 3, 4, 5] # 配列同士の連結 (上記は +=と略記可)
```

Python で文字列に対しても `+` 演算が用意されていることは既に紹介したが、`+=` という演算も用意されている。数に対する `+=` 同様に、これらも破壊的操作である。

なお、C++ の配列は、C の配列に由来するもので、Python と比べて制限が大きい。代わりに標準ライブラリの `std::vector<int>` を用いると良い。

```
C++11
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main() {
5     vector<int> a = { 1, 2, 3 };
6     for (size_t i=0; i<a.size(); ++i)
7         cout << a[i] << endl; // 1, 2, 3 が一行毎に表示される
8
9     cout << endl; // 下記と区別するための空行
10    a.push_back(4); // a に 4 を追加
11
12    for (auto n: a) // a の要素を一巡する別の記法
13        cout << n << endl; // 1, 2, 3, 4 が一行毎に表示される
14 }
```

例題

1 から  $n$  までの配列 (`iota`)

(金子)

1 から与えられた正の整数  $n$  までを順に要素として持つ配列を返す関数 `iota` を作成せよ。

```

>>> iota(5)
[1, 2, 3, 4, 5]
>>> iota(7)
[1, 2, 3, 4, 5, 6, 7]

```

ヒント: `a = []` から始めて, 順に `a` に要素を追加する.

回答例 (一部は白文字で書いてある):

```

Python3 1 def iota(n):
2     = []
3     for i in range(1, n+1):
4         .append(i)
5         # ループ不変条件は..
6     return

```

#### 例題

#### 倍数フィルタ (`filter_multiple37`)

(金子)

整数を要素とする配列 `a` に対して, 3 または 7 で割り切れる全ての数を含む配列を関数 `filter_multiple37(a)` を作成せよ.

```

>>> filter_multiple37(iota(15))
[3, 6, 7, 9, 12, 14, 15]

```

ヒント: `if` 文を用いて条件が成り立った時のみ, 答えの配列に要素を加える.

#### 問題

#### 約数列 (`factor`)

(山口和)

与えられた数  $n (> 0)$  の約数を全て求め, 配列として返す関数 `factor(n)` を作成せよ. 効率は  $n$  回の除算を行うもので良い.

#### 問題

#### たきめつきね (`decrypt2`)

(金子)

二つの文字列 `a`, `b` に対して, 両者を先頭から2文字ずつ交互に混ぜた文字列を `s` とする. 例えば, `a="ABCD"`, `b="EFGH"` の時, `s="ABEFCDDGH"` である. この `s` から逆に `a`, `b` を求めたい. 与えられた `s` に対応する, 2つの文字列を要素として持つ配列 `[a, b]` を返す関数 `decrypt2(s)` を作成せよ. なお, 文字列 `a, b` の文字数は偶数で同じとする. すなわち文字列 `s` の長さは4の倍数と仮定して良い.

```
>>> decrypt2("takinutukine") 1
['tanuki', 'kitune'] 2
```

ヒント:

```
Python3 1 def decrypt2(x):
2     a = ""
3     b = ""
4     for i in range(len(x)):
5         # i が何文字目かを考えて x[i] を a または b に加える
6         # a, b のループ不変条件は...
7     return [a, b]
```

```
C++11 1 #include <vector>
2 #include <string>
3 #include <iostream>
4 using namespace std;
5 vector<string> decrypt2(string x) {
6     string a="", b="";
7     for (size_t i=0; i<x.size(); ++i) {
8         // i が何文字目かを考えて x[i] を a または b に加える
9     }
10    return { a, b };
11 }
12 int main() {
13     auto v = decrypt2("takinutukine");
14     cout << v[0] << endl;
15     cout << v[1] << endl;
16 }
```

#### 問題

たぬき暗号 (erase\_ta)

(金子)

与えられた文字列  $x$  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  $\text{erase\_ta}(x)$  を作成せよ。文字列  $x$  は2文字以上とする。しかし, "ta" は, 偶数文字目奇数文字目いずれにも来るかもしれない。

Python で文字列  $x$  の  $i$  文字目から2文字を取り出すには,  $x[i:i+2]$  と記述する。

実は Python には `replace` という置換を行うメソッドが用意されているので, 実用にはそちらを使うべきである。この演習では, テストケースの作成に活用すると良い。

```
>>> "tatotauktayotauttao".replace("ta", "") 1
'toukyouto' 2
```

## 3.6 関数のパラメータ化 (発展)

関数を用いる際に, 計算ごとに異なる値はパラメータとして呼び出し時に与える。

多くの言語では, 関数自体をパラメータとして受け渡すことができる。大枠の計算手順の中で, 小さな計算だけが違うような場合には, その部分を表す関数をパラメータとして渡すと良い。

Python の場合, 関数を渡す場合に関数名を括弧を付けずに書き, 受け取った関数を使う場合には通常の関数と同様に実行する。(後者は, 2.4 節でも紹介した)

```
Python3 1 def f_sample(x):
2     return 3.0*x**2 + 2.0*x + 1.0
3
4 def value_at_zero_and_ten(f): # f に応じて何かをする関数
5     y0 = f(0) # f(0)
6     y10 = f(10) # f(10)
7     return [y0, y10]
```

```
>>> value_at_zero_and_ten(f_sample) 1
[1.0, 321.0] 2
```

C++11 の場合, double を引数に取り double を返す関数は, function<double(double)> という型で表す。

```
C++11 1 #include <functional>
2 #include <vector>
3 #include <iostream>
4 using namespace std;
5 double f_sample(double x) {
6     return 3.0*x*x + 2.0*x + 1.0;
7 }
8 vector<double> value_at_zero_and_ten(function<double(double)> f) {
9     double y0 = f(0);
10    double y10 = f(10);
11    return { y0, y10 };
12 }
13 int main() {
14     vector<double> v = value_at_zero_and_ten(f_sample);
15     cout << v[0] << ' ' << v[1] << endl;
16 }
```

## 問題

## 数値積分と関数パラメータ ★(integral)

(金子)

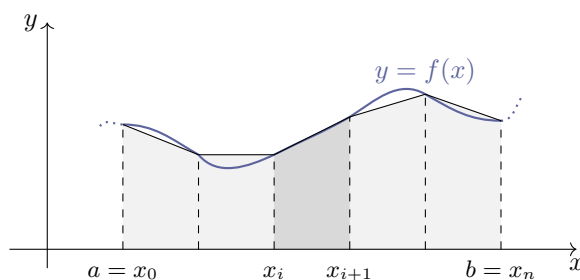
数値  $x$  を引数に取り, 対応する数値を返す関数を  $f$  とする. 区間  $[a, b]$  ( $a \leq b$ ) の定積分を, 近似して求める関数を作成せよ. 近似手法として, 台形公式とシンプソン公式の2種類を使えるようにせよ.

(この課題ではテストケースの代わりに) 様々な関数  $f$  と分割数  $n$  に応じて, 答えの精度がどのように変化するかを観察して, 仮説を立てて議論せよ. 実験結果と議論を, コメントにコピーペーストするなどしてソースコードと合わせて提出せよ.

精度については, 解析的に原始関数 ( $F(x)$ ) を求められる関数 (たとえば  $f(x) = x^2$  と  $F(x) = 1/3x^3$ ) を対象に用いて,  $F(b) - F(a)$  を真の値として差を議論する.

■台形公式 近似のもっとも簡便な方法は, 区間を細かく分けて長方形の面積の和で近似するものである.

小分けにした短冊部分について, 長方形を台形に変更すると, 近似の精度を高めることができる. すなわち, 分割数を  $n$ , 分割した各区間の幅を  $h = (b - a)/n$ ,  $x_i = a + ih$  として,  $\sum_{0 \leq i < n} h(f(x_i) + f(x_{i+1}))/2$  を答えとする. ( $\sum$  を整理すると効率が良くなるが, 定義通りに計算しても良い)



■シンプソン公式 (Simpson's rule) 直線の代わりに二次式で近似すると, さらに精度が改善する.

一般に, 3点  $(a, v_a), (b, v_b), (c, v_c)$  を通る2次の多項式は次のように一意に定まる:

$$L(x) = \frac{(x-b)(x-c)}{(a-b)(a-c)}v_a + \frac{(x-a)(x-c)}{(b-a)(b-c)}v_b + \frac{(x-a)(x-b)}{(c-a)(c-b)}v_c.$$

$a = x_i, b = x_{i+1}$  として, 区間内の三点  $(a, f(a)), (\frac{a+b}{2}, f(\frac{a+b}{2})), (b, f(b))$  を通る補間多項式を整理すると, その区間の面積は以下のように近似できる:

$$\int_a^b f(x)dx \approx \frac{b-a}{6} \left( f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right).$$

もし区間で  $f(x)$  が二次以下の多項式であれば, 計算誤差を無視すれば正確な値を得ることが出来る.

### 3.7 この章の提出課題

以下のいずれかで合格:

- 問題 (タイトル部分に ★ なし) を全て Python で解く (通常コース)



- ★つきの問題を 1 題解く (言語指定なし)  
+ 通常の問題を 1 問以上解いてループ不変条件とユニットテストを添える

提出条件:

- 各関数にテストケースが付加されていること **New!**
- ループ不変条件が明確にコメントで記述されていること **New!** (本日 3.4 節では必須, 他も適宜推奨)
- 冗長なコードがないか確認し, 標準的な書き方に整えられていること (\*) **New!**
- 指定した課題が, 指定した関数名で作成されていること
- 提出ファイルは, 1.A 節を参考にして, jupyter から作成した weekN- (学生証番号).ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中か完成品か, 授業中に既に OK をもらっていればその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など, (途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む. (何も苦労がなかった場合を除き, 自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

(\*) の項目の一つを挙げる. コード中に

```
Python3 1 if a:
        2     return True
        3 else:
        4     return False
```

という式があったら, `if` ブロックの中が評価されて値をとることを考えると, これは

```
Python3 1     return a
```

と書き換えても同じである. この書き換えは, 式 `a` の部分が, 複雑な式であっても成り立つ. したがって, この演習では後者の記法を推奨する.

## 3.A テスト駆動開発 ★

機能を拡張する場合は, (1) 先にテストケースを作る (2) すぐにテストを動かしてそのテストが働いている (i.e., 失敗する) ことを確認する (3) 関数を変更する (4) 先ほど失敗していたテストが今度は成功する (他のテストの結果は変化がない) ことを確認する, という手順を繰り返すと良いとされる. 各ステップを省略すると, テスト自体が間違っていたり, 変更により齟齬が生まれたりといったケースで, 発見が遅れうるためである.

### 3.A.1 閏年の判定

ここでは閏年を判定する関数を開発するという架空のストーリーを通して, 関数の実装とテストデータを並行して充実させることを経験する. このようにテストを作成しながら開発すると, テストを作成しない場合よりタイプする文字数は 2 倍以上になることが多いが, 一般には十分に元が取れる. 一般にバグが混入すると解決には開発時間の 10 倍以上の時間を要することが珍しくないが, ユニットテストを併用した開発は, バグの混入する機会を減少させるためである.

## 失敗するテストの作成

`is_leap_year(y)` という関数の雛形を作成する. この関数の目的は, 引数 `y` で与えられた年が閏年かどうかを真偽値で返すことである.

```
Python3 1 def is_leap_year(y):
2         return True # とりあえずなんでも閏年
```

この関数をテストするクラスを作製する.

```
Python3 1 import unittest # 定型句
2 class TestLeap(unittest.TestCase): # 定型句
3     def test_leap(self): # 名前を "test_" で始める
4         self.assertFalse(is_leap_year(2015)) # 2015 年は平年
5 unittest.main(argv=['first-arg-is-ignored'], exit=False) # jupyter
    で unittest を行う場合の定型句
```

動かすと以下のように, 失敗 (F や failures に注目) が報告される.

```
F 1
===== 2
FAIL: test_leap (__main__.TestLeap) 3
----- 4
Traceback (most recent call last): 5
  File "<ipython-input-6-cla2711afc08>", line 6, in test_leap 6
    self.assertFalse(is_leap_year(2015)) # 2015 年は平年 7
AssertionError: True is not false 8
9
----- 10
Ran 1 test in 0.002s 11
12
FAILED (failures=1) 13
```

中頃の line 6, in test\_leap から, 失敗したテストを含む関数と, 行数を読み取る. (もっとこの場合はテストが一つしか無いので自明である). このように, まず間違いようがない単純な段階でテストを意図通りに失敗させると良い. テストが実は動いていないとか, 勘違いが二重に作用してテストの不備が見過ごされるなどを防ぐため.

## 実装とテストの拡充

では, 関数 `is_leap_year(y)` を編集して, もう少しまともに閏年が4年に一度であることを判定するようにする.

```
Python3 1 def is_leap_year(y):
2     if y % 4 == 0:
3         True
4     else:
```

5                    **False**

これに伴い、テストも拡充させる。関数の中で4で割った剰余を用いているので、連続する4年間をテストすること全ての剰余のケースを試す。

```
Python3 1 class TestLeap(unittest.TestCase): # 定型句
2         def test_leap(self): # 名前を "test_" で始める
3             self.assertFalse(is_leap_year(2015))
4             self.assertTrue(is_leap_year(2016))
5             self.assertFalse(is_leap_year(2017))
6             self.assertFalse(is_leap_year(2018))
```

これを実行して、テストが全て成功することを確認する。

#### 実装の変更

ここで関数 `is_leap_year(y)` の実装が、冗長であることに気づいたとする。if 文を使う必要はないので、以下のように書き換えてみよう。

```
Python3 1 def is_leap_year(y):
2         return y % 4 == 0
```

書き換えたら必ずテストを実施し、書き換えによって失敗することがないことを確認する。

バグの原因の一つは、プログラマが等価だと思って書き換えた変更が、実際には勘違いであるケースである。このような書き換えの前後には、テストを行って関数の挙動に変化がないことを確認すると良い。

#### 仕様の変更

これまでは(ストーリー上)忘れていたが、実は「100で割り切れる年は閏年ではない」と上司の指示で(あるいはユーザからの指摘)で分かったとする。

このような場合には、関数を変更する\*前に\*テストを変更する。これまでのメソッド `test` に追加することもできるが、ここでは `test_leap100` というメソッドを新たに追加し、そこに新たに `assertTrue`, `assertFalse` 文を用いてテストを記述する。このようにクラス内にメソッドを多数記述できるので、意味のまとまり毎に分けておくと、見通しが良い。ただし、メソッド名は `test` で始める必要がある(それらのもののみ実行される)。

```
Python3 1 class TestLeap(unittest.TestCase): # 定型句
2         def test_leap(self): # 名前を "test_" で始める
3             ...
4
5         def test_leap100(self): # 新たに作成
6             self.assertTrue(is_leap_year(1896))
7             self.assertFalse(is_leap_year(1900)) # メインのテスト
8             self.assertTrue(is_leap_year(1904)) # 念のため前後の年も
```

実際にテストが失敗することを、以下のように確認する。これにより新たに作成した `test_leap100` が正常に働いていることが分かる。

```

.F
=====
1
2
3
4
5
6
7
8
9
10
11
12
13
FAIL: test_leap100 (__main__.TestLeap)
-----
Traceback (most recent call last):
  File "<ipython-input-9-40695d0915a9>", line 9, in test_leap100
    self.assertFalse(is_leap_year(1900)) # メインのテスト
AssertionError: True is not false
-----
Ran 2 tests in 0.003s

FAILED (failures=1)

```

この準備を経て、いよいよ関数を修正する。

```

Python3 1 def is_leap_year(y):
2         return not (y % 100 == 0) and y % 4 == 0

```

実際に、失敗していたテストが成功するようになったことを確認する。

例題

より正確な閏年

(金子)

現実世界では 400 で割り切れる年は 100 で割り切れる年であっても閏年である。適切なテストを追加したうえで、そのような条件を含めた `is_leap_year(y)` を作成せよ。

### 3.B C++ でのテスト

C, C++ では `assert` という標準ライブラリを用いて、同様のテストを実現可能である。演習の範囲では、`assert` で十分であろう。`assert` の中には、`true` になるはずの式を記述する。

```

C++ 1 #include ... // 自分のファイルを読み込み
2 #include <cassert> // 追加
3 int main() {
4     assert(! is_leap_year(2014));
5     assert(is_leap_year(2015)); // 失敗するテスト
6 }

```

卒業研究やプロジェクト実習などで中規模以上のプログラムを開発する場合は、あるいは実用的には、`boost::test` や `gtest` などの外部ライブラリを用いると、圧倒的に便利である。詳しくは各ライブラリのドキュメントを参照のこと。

## 第 4 章

# 参照と二次元配列

## References and two-dimensional arrays

2020-04-29 Wed

### 概要

先週に引き続き、関数内でデータを加工して返すタイプの関数の作り方を演習する。

```
def 名前(仮引数1, 仮引数2...):           1
    変数 = 初期値 # 配列や二次元の配列を扱う。           2
    for ...                               3
        変数を加工                                       4
    return 変数                                           5
```

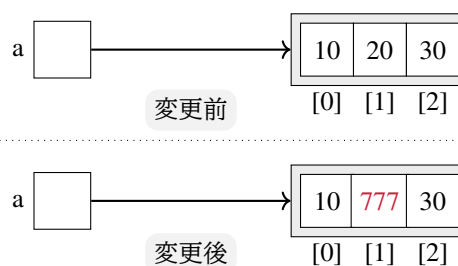
なお、先週の補足は 4.A に。

## 4.1 配列の要素の書き換え

先週導入した、代入文を拡張し配列の一部分のみを変更する文法を導入する。なお、Python では文字列をこのように変更することはできない。

```
{配列を表す式}[{整数を表す式}] = 式 1
```

```
>>> a = [10, 20, 30]
>>> a
[10, 20, 30]
.....
>>> a[1] = 777
777
>>> a
[10, 777, 30]
```



## 例題

## 7の倍数

(金子)

正の整数  $n$  が与えられる。整数  $0 < i < n$  に対して、 $i$  が7の倍数なら  $a[i]$  が `True`、 $i$  が7の倍数でなければ  $a[i]$  が `False` となるような配列を返す関数 `array7(n)` を作成せよ。なお0番目の要素は `False` とする。

要素数  $n$  で各要素の初期値が  $v$  である配列は、`[v for _ in range(n)]` で作成できる。このような表記を Python では リスト内包表記 と呼ぶ。便利なのでこの資料でも多用する。

Python3

```
1 >>> [3 for i in range(10)]
2 [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

回答例:

Python3

```
1 def array7(n):
2     a = [False for i in range(n)] # 初めに全てが False の配列を作成
3     for i in range(1, n):
4         if # i が7の倍数なら
5             a[i] = True
6         # ループ不変条件は...
7     # ここまでに7の倍数がすべてかつ7の倍数のみが True になっている
8     return a
```

## 問題

## 幸運の数 (lucky\_array)

(金子)

正の整数  $i$  が、(7の倍数であるか、または、各桁の和が7の倍数である) 時、かつその時に限り幸運であると定義する。整数  $0 < i < n$  に対して、 $i$  が幸運なら  $a[i]$  が `True`、 $i$  が幸運でなければ  $a[i]$  が `False` となるような配列を返す関数 `lucky_array(n)` を作成せよ。なお0番目の要素は `False` とする。

```
>>> lucky_array(3)
[False, False, False]
```

1  
2

幸運の判定には、5/7の資料の `digit_sum()` を組み合わせると、楽に書ける。  
テストには補助変数を用意するほうがおそらく読みやすい。

Python3

```
1 # 読みにくい
2 self.assertEqual([False, False, False, False, False, False, False, True, False, False, False],
3 # 読みやすい?
4 lucky_numbers = [False for _ in range(30)]
5 lucky_numbers[7] = True
```

```

6     lucky_numbers[14] = True
7     lucky_numbers[21] = True
8     lucky_numbers[28] = True
9     lucky_numbers[16] = True
10    lucky_numbers[25] = True
11    self.assertEqual(lucky_numbers, lucky_array(30))

```

例題

選抜

(金子)

真偽値を要素とする配列  $a$  が与えられる。  $0 \leq i < \text{len}(a)$  である整数  $i$  について、  $a[i]$  が `True` であるような  $i$  を昇順に並べた配列を返す関数 `select_index(a)` を作成せよ。

Python3

```

1 >>> select_index([False, False, True, False, True])
2 [2, 4]

```

問題

エラトステネスの篩 (`prime_array`)

(金子)

正の整数  $n$  未満の素数を配列として返す関数 `prime_array(n)` を作成せよ。

```

>>> prime_array(20)
[2, 3, 5, 7, 11, 13, 17, 19]

```

1  
2

回答方針: まずは整数  $1 < i < n$  に対して、  $i$  が素数なら  $a[i]$  が `True`、  $i$  が合成数なら  $a[i]$  が `False` となるような配列を作成する

- 初めに全ての要素を `True` とする
- $0, 1$  を `False` とする
- $2$  を残し、  $4$  以上の  $2$  の倍数を全て `False` とする
- $3$  を残し、  $6$  以上の  $3$  の倍数を全て `False` とする
- $4$  は既に `false` になっている (=合成数)。 また  $4$  の倍数も全て既に `False` になっているので何もしない
- (一般化) 「ある整数  $i$  が `True` の場合、  $i$  を残し、  $i$  より大きく  $n$  未満の  $i$  の倍数を全て `False` とする」という手順を  $i=2$  から、  $i=\text{math.floor}(\text{math.sqrt}(n))$  まで繰り返す。

## 4.2 二次元配列

Python では配列の要素に配列を配置することで、 二次元配列 に相当するデータを表現することができる。

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

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

```
>>> a[0]
```

```
[8, 3, 4]
```

```
>>> a[1]
```

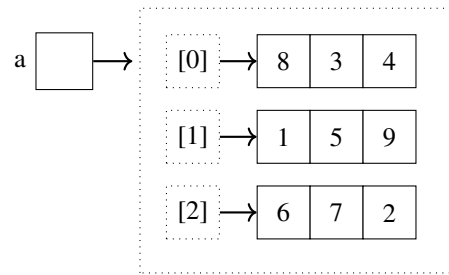
```
[1, 5, 9]
```

```
>>> a[2]
```

```
[6, 7, 2]
```

```
>>> a[1][2]
```

```
9
```



なお、縦横の概念は模式的なものであり、逆に解釈することもできる。

例題	横の和	(金子)
整数を要素とする二次元配列 $a$ と行番号 $r$ が与えられる。 $0 \leq c < \text{len}(a[r])$ の範囲の列番号 $c$ について、各要素 $a[r][c]$ の和を求める関数 $\text{rsum}(a, r)$ を定義せよ。ただし行番号 $r$ は 0 以上、 $\text{len}(a)$ 未満の整数とする。		

```
>>> a = [[1,2,3], [10,20,30]]
```

```
1
```

```
[[1, 2, 3], [10, 20, 30]]
```

```
2
```

```
>>> rsum(a,0)
```

```
3
```

```
6
```

```
4
```

```
>>> rsum(a,1)
```

```
5
```

```
60
```

```
6
```

例題	縦の和	(金子)
整数を要素とする二次元配列 $a$ と列番号 $c$ が与えられる。 $0 \leq r < \text{len}(a)$ の範囲で、各要素 $a[r][c]$ の和を求める関数 $\text{vsum}(a, c)$ を定義せよ。ただし $a$ の列数は全ての行で等しく、列番号 $c$ は 0 以上かつ列数未満の整数とする。		

```
>>> vsum(a,0)
```

```
1
```

```
11
```

```
2
```

```
>>> vsum(a,1)
```

```
3
```

```
22
```

```
4
```

```
>>> vsum(a,2)
```

```
5
```

```
33
```

```
6
```

二次元配列 (height 行, width 列, 初期値 initial) を作成するには、さしあたり以下の関数 `make2d` を用いる。



```
Python3 1 def make2d(height,width):
2     a = [[0 for _ in range(width)] for _ in range(height)]
3     return a
```

問 (発展):

make2d に代えて, `[[0]*width]*height` を用いたとする. どのような差異があるかを考察せよ.  
(考え方と回答は 4.4 節で)

C++ の場合は `vector` の入れ子を用いると良い:

```
C++11 1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 int main() {
5     vector<vector<int>> a = {{8,3,4}, {1,5,9}, {6,7,2}};
6     for (auto& row: a) {
7         for (auto e: row)
8             cout << e << ' ';
9         cout << endl;
10    }
11 }
12 /* 実行結果
13 8 3 4
14 1 5 9
15 6 7 2
16 */
```

例題

2 倍

(金子)

縦横の行数列数が揃った整数を要素とする二次元配列 `a` が与えられるので, 同じ行数列数を持ち, 各要素が 2 倍になっている行列を返す関数 `matrix_double(a)` を作成せよ.

```
>>> matrix_double(a) 1
[[2, 4, 6], [20, 40, 60]] 2
```

回答例

```
Python3 1 def matrix_double(a):
2     b = make2d(len(a), len(a[0])) # 答えとなる配列
3     for r # 適切な範囲の r 全てについて
4         for c # 適切な範囲の c 全てについて
5             b[r][c] = a[r][c]*2
6     return b
```

問題	表計算 (extend_sum)	(AOJ ITP)
<p>二次元配列が与えられるので、縦方向と横方向に両方にその和を加えた二次元配列を (新しく作成して) 返す関数 <code>extend_sum(a)</code> を作成せよ。 (元の配列は変更しないとする)</p> <p>入力の二次元配列は、長方形型 (どの列の行数もどの行の列数も同じ) とする。</p> <p>図解 <a href="http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_7_C">http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ITP1_7_C</a></p> <p>(リンク先では表示するよう指示があるが、この演習では配列を返す関数を作成すること)</p>		

```

>>> a
[[1, 2, 3], [10, 20, 30]]
>>> extend_sum(a)
[[1, 2, 3, 6], [10, 20, 30, 60], [11, 22, 33, 66]]

```

### 4.3 有限ライフゲーム

ライフゲーム (Conway's Game of Life) では無限に広がる二次元グリッドに配置された各マスについて、生物が生きているか死んでいるかの 2 状態を扱う。この演習では有限のフィールドに限定し、生死をそれぞれ `True, False` で表すことにする。

例題	初期状態作成	(金子)
<p>二次元配列の大きさ <code>height, width</code> と、生きているマスの一覧を表す配列 <code>living</code> が与えられる。各 <math>0 \leq i &lt; \text{len}(\text{living})</math> に対して、<code>living[i][0]</code> 行 <code>living[i][1]</code> 列目が生きているマスとする。生きているマスが <code>True</code> そうでないマスが <code>False</code> を持つ二次元配列を返す関数 <code>make_field(height,width,living)</code> を作成せよ。</p>		

```

>>> make_field(2,3,[])
[[False, False, False], [False, False, False]]
>>> make_field(1,3,[[0,0],[0,1]])
[[True, True, False]]

```

ライフゲームの初期状態には、何ステップか経てやがて死滅するもの、やがて初期状態に戻り周期的に繰り返すもの、無限に増え続けるものなど様々なパターンがあることがわかっている。その中で移動しながら変形を繰り返す移動型の一つである、グライダーと呼ばれる初期状態を作成し、図示してみよう。

`matplotlib` というライブラリを用いる。

```

Python3 1 %matplotlib notebook

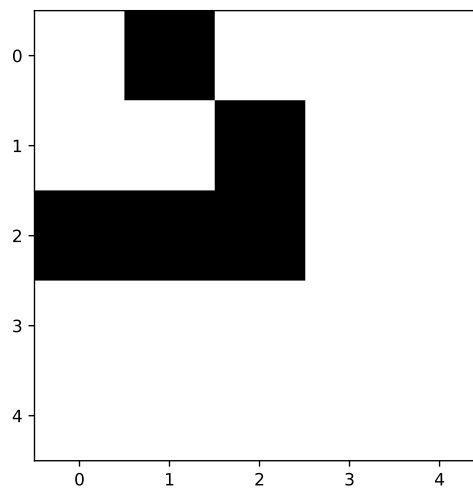
```

```

2 import matplotlib
3 import matplotlib.pyplot as plt
4
5 def plot(array2d):
6     fig = plt.figure()
7     ax = fig.add_subplot(1,1,1)
8     im = ax.imshow(array2d, interpolation="nearest", cmap="Greys")
9     fig.show() # fig.savefig("plot.pdf")
10
11 a = make_field(5, 5, [[0,1], [1,2], [2,0], [2,1], [2,2]])
12 plot(a)

```

この plot 関数を、そのまま使って良い。<sup>\*1</sup>



各  $a[r][c]$  について 0 が白に、1 が黒に対応し、位置は左上が原点で  $r$  が 1 増えると 1 行下の行が、 $c$  が 1 増えると 1 列右の列が対応する

ライフゲームでは次の時刻での各マスの生死は、現在のマスの状況と周囲 8 近傍の生きているマスの数に基づいて、次の規則により決定される。この規則に基づき次の時刻の状態を求める関数を順に作成しよう。

現在	8 近傍の生のマス			
	0,1	2	3	4-8
生	死	生	生	死
死	死	死	生	死

<sup>\*1</sup> なお、全ての要素が True の配列を表示すると真っ白になるという“仕様”がある。imshow が値を正規化するため。

## 例題

## 範囲内の座標

(金子)

長方形のフィールドを表す二次元配列 `field` と、`r` 行 `c` 列目を表す整数 `r`, `c` が与えられる。座標がフィールド内部かどうかを真偽値で返す関数 `inside(field, r, c)` を作成せよ。整数 `r`, `c` は配列の外側であったり、負である場合もある。

```
>>> inside(make_field(1,2,[]), 0,0) 1
True 2
>>> inside(make_field(1,2,[]), 0,1) 3
True 4
>>> inside(make_field(1,2,[]), 0,2) 5
False 6
>>> inside(make_field(1,2,[]), 0,-1) 7
False 8
```

## 例題

## 座標のマス

(金子)

指定されたマスの生死を整数で返す関数 `is_living_cell(field, r, c)` を作成せよ。生を 1, 死を 0 とする。ただし、指定されたマスがもしフィールドの外であれば、死と扱う。(引数の意味は前問と同じである)

## 例題

## 8 近傍の生きているマスの数

(金子)

(引数の意味は前問と同じである) 指定されたマスの 8 近傍で生きているマスの数を数える関数 `count(field, r, c)` を作成せよ。ただし、8 近傍とは斜めを含めた隣のマスであり、 $(r', c')$  が  $(r, c)$  の 8 近傍であれば(かつその時に限り)、 $|r' - r| \leq 1$ ,  $|c' - c| \leq 1$ ,  $(r', c') \neq (r, c)$  が成り立つ。

Python3

```
1 for i in [-1,0,1]:
2     for j in [-1,0,1]:
3         if (i != 0 or j != 0):
4             # r+i, c+j が r, c の 8 近傍のどこか
```

## 例題

## 次の時刻の生死

(金子)

ある特定のマスについて、現在の生死を表す整数 `living` と現在の 8 近傍で生きているマスの数を 0 以上 8 以下の整数で表す `neighbor_count` が与えられる。それらを元に、そのマスの次

(例題 続き)

の時刻の生死を求める関数 `next_cell_living(living, neighbor_count)` を作成せよ。

問題

次の状態 (`next_field`)

(金子)

真偽値 (True または False) の要素の長方形の領域を表す二次元配列 `a` を与えられて、ライフゲームのルールに従った次の状態を返す関数、`next_field(a)` を作成せよ。

```
>>> a
[[False, True, False, False, False], [False, False, True, False, False],
 [True, True, True, False, False], [False, False, False, False, False],
 [False, False, False, False, False]]
>>> next_field(a)
[[False, False, False, False, False], [True, False, True, False, False],
 [False, True, True, False, False], [False, True, False, False, False],
 [False, False, False, False, False]]
>>> plot(next_field(a))
```

Wikipedia [http://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life#Examples\\_of\\_patterns](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Examples_of_patterns) や Wiki [https://www.conwaylife.com/wiki/Main\\_Page](https://www.conwaylife.com/wiki/Main_Page) などを参考に、固定パターンと、周期が2の小さなパターンを一つずつ試してみよう。

■連続アニメーション これまでは1枚1枚の絵を `show` で別々に表示してきた。Python の場合、`matplotlib` でアニメーション作成機能がある。下記のサンプルコードでは、`show_animation` でアニメーションを表示している。

```
Python3 1 %matplotlib notebook
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import matplotlib.animation as animation
5
6 def show_animation(initial_field, step):
7     a = []
8     fig = plt.figure()
9     field = initial_field
10    for i in range(step):
11        a.append([plt.imshow(field)])
12        field = next_field(field)
13    anim = animation.ArtistAnimation(fig, a, interval=500)
14    plt.show()
15    return anim
```

```
16 glider = [[0,1],[1,2],[2,0],[2,1],[2,2]]
17 show_animation(make_field(15,15,glider),50)
```

セルを実行すると、ノートブック上でアニメーションが再生される。

問題	移動型	(金子)
<p>Glider や Lightweight spaceship など移動型を動かし、周期や移動幅を調査してみよう。各型ごとに、初期配置と自分の選んだ特徴的な時刻の合計二つの時刻のフィールドの状況に関するテストケースを付与すること。さらに、アニメーション (mp4 など) を作成して、合わせて提出せよ。</p>		

アニメーションの保存は、以下のように save を呼ぶと、jupyter を実行しているディレクトリにファイルが保存される。

```
Python3 1 anim = show_animation(make_field(15,15,glider),50)
2 anim.save("filename.mp4")
```

eccs の iMac 端末では既に設定がされているが、自分の PC で実行するためには、事前に準備が必要な場合がある<sup>\*2</sup>。

問題	様々なパターン *	(金子)
<p>Glider や Lightweight spaceship など移動型や、循環するもの、射出型など様々なパターンをたくさん (5 種類以上) 動かして、周期や移動幅を調査してみよう。各型ごとに、初期配置と自分の選んだ特徴的な時刻の合計二つの時刻のフィールドの状況に関するテストケースを付与すること。さらに、アニメーション (mp4 など) を作成して提出せよ。</p>		

## 4.4 実体と参照 \*

変数とは何か、最初の説明として「データの入れ物の変数である」などと説明されるが、入れ物の中に何が入っているか、複雑なプログラムを書く際は実体と参照の区別が必要である。

■参照の複製 次の二つの例で数と配列の違いを見てみよう：

<sup>\*2</sup> 本教材では設定方法をサポートしないが、以下の動作報告があるので紹介する。Mac の場合、brew install ffmpeg とする。MS Windows の場合は、ffmpeg を適切な方法でインストールしたうえで、plt.rcParams['animation.ffmpeg\_path'] = "path-to-ffmpeg" と Writer = animation.FFMpegWriter() というコードを保存の前に追加する。

```

>>> a = 3
3
>>> b = a
3
.....
>>> a += 100
103
>>> a
103
>>> b
3

```

整数の場合は、変数には値そのものが格納されている。したがって `b=a` とした後で、`a` の値を変更しても、`b` の値は変わらない。一方、配列の場合は、配列の実体はヒープ領域と呼ばれる別の場所に確保され、矢印で図示した参照によって実体と変数と関連付けられる。矢印の先を参照先、元を参照元と呼ぶ。そして、`b=a` とした際には、参照のみがコピーされるため、参照先の要素を変更した場合は、`a, b` 両方から変更後の内容が参照される。

```

>>> a = [3,4,5]
[3, 4, 5]
>>> b = a
[3, 4, 5]
.....
>>> a.append(100)
[3, 4, 5, 100]
>>> a
[3, 4, 5, 100]
>>> b
[3, 4, 5, 100]

```

参照ではなく、実体を複製したい場合は、新しい配列をリスト内包表記や `[]` で構築して各要素を自分でコピーする、もしくは `copy.deepcopy` というメソッドを用いる。

```

>>> a = [3,4,5]
[3, 4, 5]
>>> import copy
>>> b = copy.deepcopy(a)
[3, 4, 5]
.....
>>> a.append(100)
>>> a
[3, 4, 5, 100]
>>> b
[3, 4, 5]

```

■関数呼び出しと参照 Python や C++ では関数呼び出しの際に引数に、値が複製される。これを 値渡し と呼ぶ。

```

Python3 1 def add3(n):
        2     n += 3

```

```

3      return n

```

---

```

>>> n = 5
5
>>> add3(n)
8
>>> n
5
5 # n の値は元のまま

```

すなわち関数内部で引数の値を変更しても、関数の呼び出し元の変数の値には影響がない。しかしながら、引数が参照である場合には、複製された参照は同じ参照先を指すために、参照先の変更は関数の呼び出し元からも可視である。

```

Python3 1 def add3_array(a):
2     a.append(3)
3     return a

```

---

```

>>> a = [1,2]
>>> add3_array(a)
[1, 2, 3]
>>> a
[1, 2, 3] # a の指す配列は変更されている

```

多くのプログラミング言語で、型によって実体を扱うか参照を扱うかの区別がある。特に Java では primitive type (整数など) と reference type (String など) と明確に区別される<sup>\*3</sup>。Python で変数の型は type 関数により知ることが出来る。Python の場合、厳密には全ての変数は object への参照を持つが、immutable (変更不可) な型の場合は (int, float, str, tuple, etc.), 他の言語で実体を扱う場合と同等に考えることができる。mutable (変更可能) の型 (list, etc.) の場合は、参照の扱いに関する上記の注意が必要である。

```

>>> a = 1
>>> type(a)
<class 'int'>
>>> b = 1.0
>>> type(b)
<class 'float'>
>>> c = [1, 2, 3]
>>> type(c)
<class 'list'>

```

■二次元配列と参照 Python では配列の配列として、二次元配列を実現する。

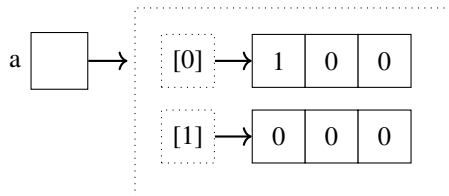
<sup>\*3</sup> <https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html>



```

>>> a = make2d(2,3)
>>> a[0][0] = 1
>>> a
[[1, 0, 0], [0, 0, 0]]

```

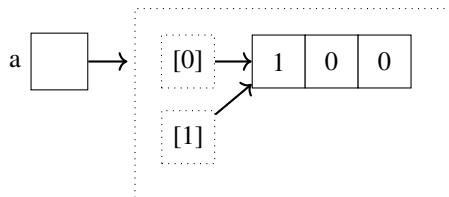


失敗例:

```

>>> a = [[0]*3]*2
>>> a
[[0, 0, 0], [0, 0, 0]] # 一見, 二次元配列
>>> a[0][0] = 1
>>> a
[[1, 0, 0], [1, 0, 0]] # 2箇所変更されている

```



■C++ での参照 C++ では、参照を扱う場合は、記号&を用いて明示的に記述する。Python や Java ではメモリブロックの寿命を管理する GC (ガーベージコレクション) の仕組みを持つが、C++ は GC の機構を持たないため変数のスコープと寿命をより注意深く理解する必要がある (この点は後日に譲る)。

```

C++
1 #include <string>
2 #include <iostream>
3 using namespace std;
4 void decorate(string name) { // name は複製
5     name += "-san";
6 }
7 void decorate_ref(string& name) { // name は参照
8     name += "-san";
9 }
10 int main() {
11     {
12         string name = "Taro";
13         string copy = name;
14         copy[0] = 'J';
15         copy[1] = 'i';
16         cout << name << endl; // Taro

```

```
17     cout << copy << endl; // Jiro
18 }
19 {
20     string name = "Taro";
21     string& copy = name; // name を参照
22     copy[0] = 'J';
23     copy[1] = 'i';
24     cout << name << endl; // Jiro
25     cout << copy << endl; // Jiro
26 }
27 {
28     string name = "Taro";
29     decorate(name);
30     cout << name << endl; // Taro
31 }
32 {
33     string name = "Taro";
34     decorate_ref(name);
35     cout << name << endl; // Taro-san
36 }
37 }
```

---

## 4.5 この章の提出課題

以下のいずれかで合格:

- 問題 (タイトル部分に ★ なし) を全て Python で解く (通常コース)
- ★ つきの問題を 1 題以上解く

提出条件:

- 各関数に対応するテストケースが記述されていること (ユニットテストは、テンプレートの末尾の class にまとめて一箇所に記述する。)
- ループ不変条件が明確にコメントで記述されていること (本日 4.1 節では必須、他も適宜推奨)
- 冗長なコードがないか確認し、標準的な書き方に整えられていること
- 指定した課題が、指定した関数名で作成されていること
- 提出ファイルは、1.A 節を参考にして、[jupyter から作成した](#) weekN-(学生証番号).ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中で完成品か、授業中に既に OK をもらっていればその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など、(途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む。 (何も苦労がなかった場合を除き、自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

## 4.A 先週の補足

■「たぬき暗号」 もし「"ta"ぬき暗号」ではなく、「"t"ぬき暗号」であれば下記のように簡単に作成可能であろう。実は"ta"を除く場合も、`if` 文の条件式を変更するだけで対応できる (考えてみよう)。

```
Python3 1 def erase_t(x):
2     s = ""
3     for i in range(len(x)):
4         if # x[i] が "t" でなければ s に追加
5             s += x[i]
6         # ループ不変条件: s は、x の先頭から i 文字目までの、t 以外の文字列に相当
7     return s
```

■どのようなテストケースを作るべきか 作成したテストケースが十分かどうか心配というコメントがあった。現状ほぼ問題ないが、不足があれば個別に指摘する。

一般的な指針は以下のとおりである:

- 標準的な値とその周辺 (例: 閏年判定なら閏年とその  $\pm 1$  年)
- 条件判断 (`if`) があれば、条件が成り立つ場合と成り立たない場合
- 引数の仕様の範囲での極端な値 (整数であれば、最大値、最小値、0、文字列であれば空文字列 "", 空の配列など)
- ループ (`for`, `while` など) があれば、ループの最小回数や最大回数を与える入力 (分析可能な場合)

なお、この演習では手動でテストケースを作成し、明らかな間違いに速やかに気づくことを目的としている。実用的には必要に応じて、自動的なテストも併用される。入力として与えられる種類が小さければ全ての場合について入出力を網羅できる場合もある。またそうでなくても、可能な範囲のデータで耐久テストをすることもある (例: 将棋の全ての盤面をテストすることはできないが、流通している棋譜に現れる局面では正常に動作する)。

異常な入力 (「整数  $n$  に対して」と問題文に書いてあるのに文字列を与えるようなケース) に対しては、煩雑を避けるためにこの演習では (特に指示がなければ) 対処不要とする。

## 第 5 章

# 再帰

*Recursion*  
2020-05-13 Wed

### 概要

再帰について演習する。線形の再帰 (for で等価な内容を書ける) から始めて、徐々に複雑なものを扱う。複雑な再帰は (簡単には) 反復ではかけないので、再帰で綺麗に書けるような処理の手順が存在することとその書き方について理解しよう。

5.1-5.3 の問題は、変数の破壊的代入を用いずに (i.e., 定数変数のみを用いて), 解くこと。

この章も、[テストケースを提出に添える](#)こと。

## 5.1 帰納的定義と線形再帰

「何かを繰り返して計算したい」状況を考える。while や for のような反復・ループによって書く方法と、関数の中から自分自身を呼び出す[再帰](#) (recursion) によって計算する方法がある。ここでは後者について掘り下げる。

例として、 $1 + 2 + \dots + n$  に相当する関数を、[漸化式](#)で定義してみよう。この関数を  $\text{sum}(n)$  と書くことにすると、次のように帰納的に定義できる。

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

この定義の中では  $\text{sum}$  自身を使っていることに注意。これは、ほとんどそのまま Python や C++ の定義に置き換えることができる:

```
Python3 1 def sum1(n):
2     if n == 1:
3         return 1
4     else:
5         return n + sum1(n-1)
```

例題	階乗 (再帰)	(増原)
正の整数 $n$ についての階乗 ( $n! = 1 \times 2 \times \cdots \times n$ ) の帰納的な定義と関数 <code>factorial(n)</code> .		

例題	倍数の個数 (再帰)	(増原)
<p>「1 から正の整数 <math>n</math> までの中にある正の整数 <math>m</math> の倍数の個数」を、再帰で数える関数 <code>count_multiples(n, m)</code>. この関数を <math>f(n, m)</math> とすると、帰納的には次のように定義できる.</p> $f(n, m) = \begin{cases} 0 & n = 0 \text{ の場合} \\ 1 + f(n-1, m) & n \text{ が } m \text{ の倍数の場合} \\ f(n-1, m) & \text{それ以外} \end{cases}$		

■末尾再帰 (tail recursion) 式を少し変形した、次の定義の  $\text{sum}'(s, n)$  を考える.

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

上記の定義中の  $s$  は、「これまでの和」に相当すると考えて、 $\text{sum}(n) = \text{sum}'(0, n)$  であることを確認せよ.

```

Python3 1 def sum2(s, n):
2     if n == 1:
3         return s+1
4     else:
5         return sum2(s+n, n-1)

```

返り値として自分自身を呼ぶ構造を末尾再帰と呼ぶ. このように書いておくと、コンパイラが最適化しやすいというメリットがある.

また、 $s$  のような補助的な変数を導入すると再帰の見通しがよくなることがある.

## 5.2 範囲を絞る: 二分法・二分探索

例題	最大公約数	(山口和)
与えられた2つの数 $a, b$ の最大公約数を求める関数 <code>gcd(a, b)</code> を作成せよ. ただし $a > b \geq 1$ とする.		

$$\text{gcd}(a, b) = \begin{cases} b & b \text{ が } a \text{ を割り切るとき} \\ \text{gcd}(b, a \% b) & \text{それ以外} \end{cases}$$

## 例題

## 平方根

(情報)

再帰を用いて、与えられた数  $x$  の平方根である  $\sqrt{x}$  を求める関数 `sqrt(x, a, b)` を作成せよ。

与えられた数  $x$  の平方根である  $\sqrt{x}$  を求める関数 `sqrt(x, a, b)` を考える。予め区間  $[a, b]$ ，すなわち  $a < x < b$  に解があることが分かっているとする。すると、 $\sqrt{x}$  の単調性から  $m = \frac{a+b}{2}$  として

$$\text{sqrt}(x, a, b) = \begin{cases} m & |m^2 - x| < \epsilon \text{ のとき} \\ \text{sqrt}(x, a, m) & m^2 > x \text{ のとき} \\ \text{sqrt}(x, m, b) & \text{それ以外} \end{cases}$$

ただし、 $\epsilon$  は許容誤差を表す。たとえば 0.0001 を用いる。また正しく計算するには  $a^2 < x < b^2$  となるような  $a, b$  を与える必要がある。

実行例:

```
>>> sqrt(3, 0, 10) 1
1.732025146484375 2
```

実装例

Python3

```
1 Epsilon = 0.0001
2 def sqrt(x, a, b):
3     m = (a+b)/2.0
4     # print(x, a, b, m) # どのような引数で呼び出されているか観察できる
5     if abs(x - m*m) < Epsilon:
6         return m
7     elif m*m > x:
8         return sqrt(x, a, m)
9     else:
10        return sqrt(x, m, b)
```

浮動小数演算の誤差を含むので、自動テストの際は `assertEqual` の代わりに `assertAlmostEqual` (`first, second, delta`) を用いて、許容誤差 `delta` を指定する<sup>\*1</sup>。

使用例

Python3

```
1 Epsilon = 0.0001
2 self.assertAlmostEqual(1.4142, sqrt(2, 0, 10), delta=Epsilon)
```

<sup>\*1</sup> <https://docs.python.jp/3/library/unittest.html#unittest.TestCase.assertAlmostEqual>

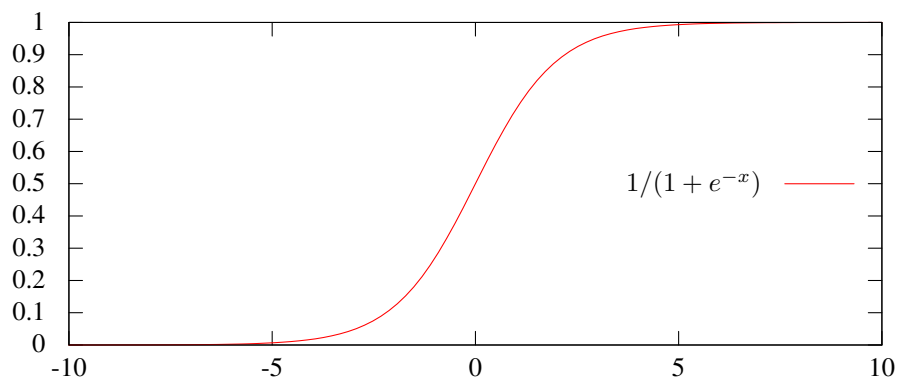
## 問題

シグモイド関数 (`sigmoid_inverse`)

(情報)

シグモイド関数  $f(x) = \frac{1}{1+e^{-x}}$  は、 $x \in [-\infty, \infty]$  を  $[0, 1]$  にマップする。再帰を用いて、与えられた数  $y$  と区間  $[a, b]$  に対して  $f(x) = y$  となるような  $x$  を求める関数 `sigmoid_inverse(y, a, b)` を作成せよ。

なお  $e^x$  は `math.exp(x)` で計算できる (`import math` を一度行う)。



## 実行例

```
>>> sigmoid_inverse(0.5, -10, 10)      1
0.0                                     2
>>> sigmoid_inverse(0.2, -10, 10)      3
-1.38671875                             4
>>> sigmoid_inverse(0.1, -10, 10)      5
-2.197265625                             6
>>> sigmoid_inverse(0.9, -10, 10)      7
2.197265625                             8
```

— maximum recursion depth exceeded —

再帰関数は、どこかで止まるように実装する必要がある。たとえば以下の関数は止まるところが定義されていない。

```
Python3 1 def inf(a):
        2     return 1+inf(a)
```

この関数を `inf(0)` と呼び出すと、

```
>>> inf(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in inf
  File "<stdin>", line 2, in inf
  File "<stdin>", line 2, in inf
  [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。

`sigmoid_inverse(y, a, b)` を作っていてこのエラーが出た場合は、

などを確認

する。確認には、`a, b, f(a), f(b), y` などを関数の入り口で表示すると良い。

#### 問題

#### 二分探索 (binary\_search)

(金子)

配列  $a$  の要素は整数で昇順に整列されているとする。添字の範囲  $[l, r)$  の  $a$  の要素に整数  $x$  と等しいものが存在すればその添字、そうでなければ  $-1$  を返す関数 `binary_search(a, x, l, r)` を、再帰を用いて作成せよ。なお探す範囲は半開区間であり、 $l \leq i < r$  で  $a[i] == x$  となるような  $i$  を探す。また正しく計算するために  $a[l] \leq x < a[r]$  かつ  $0 \leq l < r \leq \text{len}(a)$  となるような  $l, r$  を与えるとする。配列  $a$  の要素は互いに異なると仮定して良い。

なお、変数名  $l, r$  は `left` と `right` からとられている。

簡単に `binary_search` を `bs` と略記すると、帰納的定義を配列の要素の単調性から次のように書ける:

$$\text{bs}(a, x, l, r) = \begin{cases} l & l+1 = r \text{ かつ } a[l] = x \text{ のとき} \\ -1 & l+1 = r \text{ かつ } a[l] \neq x \text{ のとき} \\ \text{bs}(a, x, l, m) & l+1 < r \text{ かつ } a[m] > x \text{ のとき, 但し } m = \lfloor (l+r)/2 \rfloor \\ \text{bs}(a, x, m, r) & \text{それ以外} \end{cases}$$

注: Python3 で  $m = (l+r)/2$  を計算するためには、`//` 演算子を用いると良い。



```

>>> a=[1,3,5,7]
[1, 3, 5, 7]
>>> binary_search(a, 1, 0, len(a))
0
>>> binary_search(a, 3, 0, len(a))
1
>>> binary_search(a, 5, 0, len(a))
2
>>> binary_search(a, 7, 0, len(a))
3
>>> binary_search(a, 2, 0, len(a))
-1
>>> b=[0,1,10,100,1000]
>>> binary_search(b, 1, 0, len(a))
1

```

辞書や名簿のように順番に並んでいるデータ列にある要素が存在するかどうかを判定する問題も、同様の考え方で解くことができる。たとえば“Moore”という姓を、全体が 1024 ページの名簿から探す場合に、1024 ページの真ん中の 512 ページから始める。そのページが“M”より後なら、前を調べる。“M”以前なら、後を調べる。いずれの場合でも、探す範囲を半分に絞ることが出来るといった具合である。この探し方は、1 ページ目から順に 2,3 ページと一ページずつ最終ページまで名簿を探すより、速い。名簿のページ数を  $n$  とすると、調べるページ数は  $O(n)$  と  $O(\log_2(n))$  の差がある。

### 5.3 枝分かれを伴う再帰

ここまで見た再帰呼び出しでは、1 つの関数からその関数を 1 回だけ呼び出していた。今節以降は、複数の再帰を呼び出す場合を扱う。

#### 例題

#### フィボナッチ数

(増原)

フィボナッチ数列 (Fibonacci numbers) とは、

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

のように、「前 2 つの数の和」で作られる数の列である。 $n$  番目のフィボナッチ数  $fib(n)$  を帰納的に定義せよ。また、関数  $fib(n)$  を定義せよ。

Python3

```

1 def fib(n):
2     print("fib", n) # 関数呼び出しの際に引数を表示
3     if n == 0:
4         return 0
5     elif n == 1:
6         return 1

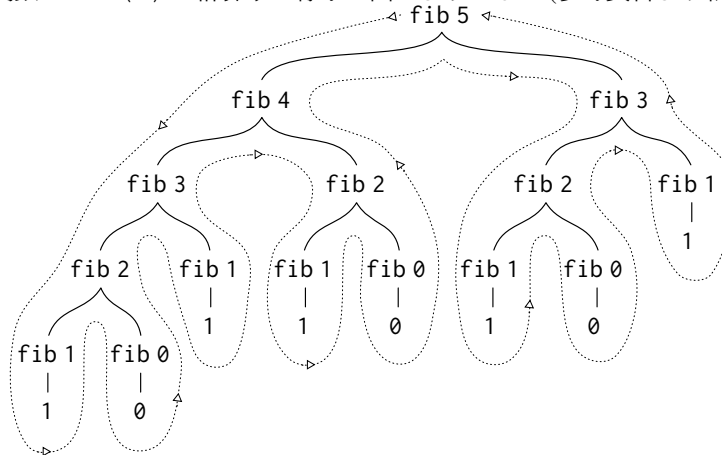
```

```

7     else:
8         return fib(n-2)+fib(n-1)

```

再帰的に定義した関数で `fib(5)` を計算する様子は図のようになる (参考資料より転載)。



なお再帰は fibonacci 数を計算することに関しては最善の方法ではない。より効率良くフィボナッチ数を求める方法は別に扱う。

例題	m 進数	(金子)
整数 $m, n$ が与えられる ( $1 \leq n \leq 8, 1 < m \leq 10$ ). $n$ 桁の $m$ 進数を小さい順に全て表示する関数 <code>mdigit(m, n)</code> を作成せよ。		

```

>>> mdigit(3,2)
00
01
02
10
11
12
20
21
22
1
2
3
4
5
6
7
8
9
10

```

簡単のため  $m = 2$  すなわち 2 進数を考える。正の整数  $n$  に対して目的を達する手続きを、 $n - 1$  に対して同様の処理を行う手続きを元に組み立てたい。つまり、 $n - 1$  桁の 2 進数を全部表示する関数を誰かが作ってくれていたとして (\*1), それを元に  $n$  桁の  $m$  進数を全部表示する関数を作れるだろうか。

$n$  桁の 2 進数は左端の 1 文字 (0 または 1) と残り  $n - 1$  桁の 2 進数に分解できる。そこで、左端に 0 を追加して  $n - 1$  桁の 2 進数を全部表示する、左端に 1 を追加して  $n - 1$  桁の 2 進数を全部表示する、というようなことをしたい。

上記を踏まえて (\*1) の関数を微調整して、左端に書いてほしい文字列 `prefix` を引数に追加し、引数 `prefix`

と  $n$  に対して、「 $n$  桁の 2 進数を全て生成して、それぞれに prefix をつけて表示する」という関数を  $B(\text{prefix}, n)$  とする。

$$B(\text{prefix}, n) = \begin{cases} \text{prefix を表示} & (n = 0) \\ B(\text{prefix} + 0, n - 1) \text{ と} \\ B(\text{prefix} + 1, n - 1) \text{ を実行} & (n > 1) \end{cases} \quad (5.1)$$

prefix	3	2,1	
10	0	00	← B(100,2) を通じて実現
10	0	01	
10	0	10	
10	0	11	
10	1	00	← B(101,2) を通じて実現
10	1	01	
10	1	10	
10	1	11	

$B(10, 3)$  の動作:  $B(100, 2)$  と  $B(101, 2)$

$B(0, n)$  が、 $n$  桁の 2 進数を全て表示することを確認せよ。2 進数だけでなく  $m$  進数に対応するには、0 と 1 のみ分岐させていた部分を、for 文などで 0 から  $m - 1$  まで分岐させれば良い。

prefix は文字列 (str) を想定して説明したが、int で表現しても良い。その場合、たとえば、prefix の右に 1 を加える操作は、 $\text{prefix} * 10 + 1$  となる。

文法: Python で、整数  $a$  を  $N$  桁で表示し、桁が足りない際に 0 を補うには以下のように行う。

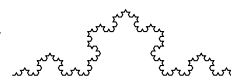
```
Python3 1 >>> print("{:05d}".format(3))
2 00003
```

## 問題

## Recursion / Divide and Conquer - Koch Curve (koch)

(AOJ)

フラクタルの一つであるコッホ曲線の頂点の列を計算して配列として返す関数  $\text{koch}(x_0, y_0, x_1, y_1, n)$  を作成せよ。ただし点  $(x_0, y_0)$  と  $(x_1, y_1)$  を始点と終点とする。また再帰の深さを  $n$  とする。



テスト用のデータ例: [http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_C&lang=ja](http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_C&lang=ja)  
(下記を参考に、頂点列を図示すること) koch

ループを用いずに再帰だけで書くことができる。また引数の破壊的変更も不要である。

```
Python3 1 # (x0, y0) から (x1, y1) までのコッホ曲線の頂点列を返す
2 # ただし (x1, y1) は含まない
```

```

3 def koch(x0,y0, x1,y1, n):
4     if n == 0:
5         return [[x0,y0]] # (x0,y0)..(x1,y1) の直線
6     else:
7         # 変数の準備
8         # 再帰 ... 部分は適当に埋める
9         return koch(x0,y0, ..., n-1) + koch(..., n-1) \
10              + koch(..., n-1) + koch(..., x1,y1, n-1)

```

ヒント: 基準線が  $(x_0, y_0)$  から  $(x_0 + dx, y_0 + dy)$  に引かれるとき, 左手側にこぶを作るには, 中点  $(x_0 + dx/2, y_0 + dy/2)$  から,  $(-dy, dx)$  方向に適当な距離を (i.e., ) 伸ばせば良い.

■matplotlib による描画 作成した関数の結果を描画してみよう. 関数の返り値  $a$  が  $[[0, 0], [100, 0]]$  のような点を表す配列の配列だった.

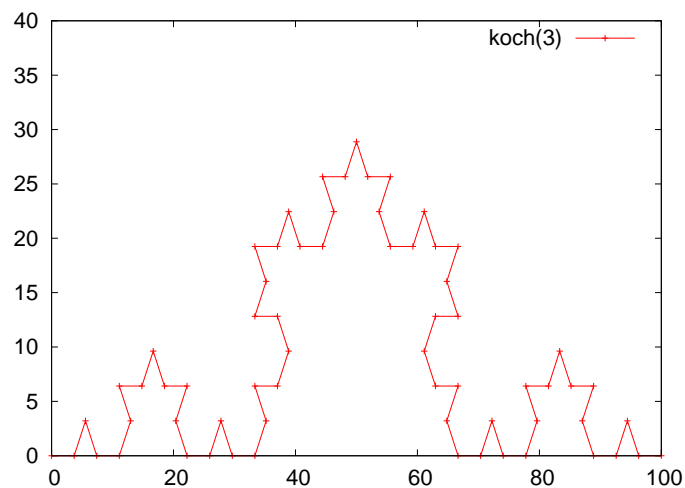
Python3

```

1 %matplotlib inline
2 def plot_koch(a):
3     import matplotlib.pyplot as plt
4     import numpy as np
5     fig = plt.figure()
6     ax = fig.add_subplot(1,1,1)
7
8     a_ = np.array(a)
9     xlist = a[:, 0]
10    ylist = a[:, 1]
11
12    ax.plot(xlist, ylist)
13    ax.set_title('koch_curve')

```

関数 `plot_koch` にこのような配列の配列を与えると, jupyter notebook 上に結果を描画することが出来る.

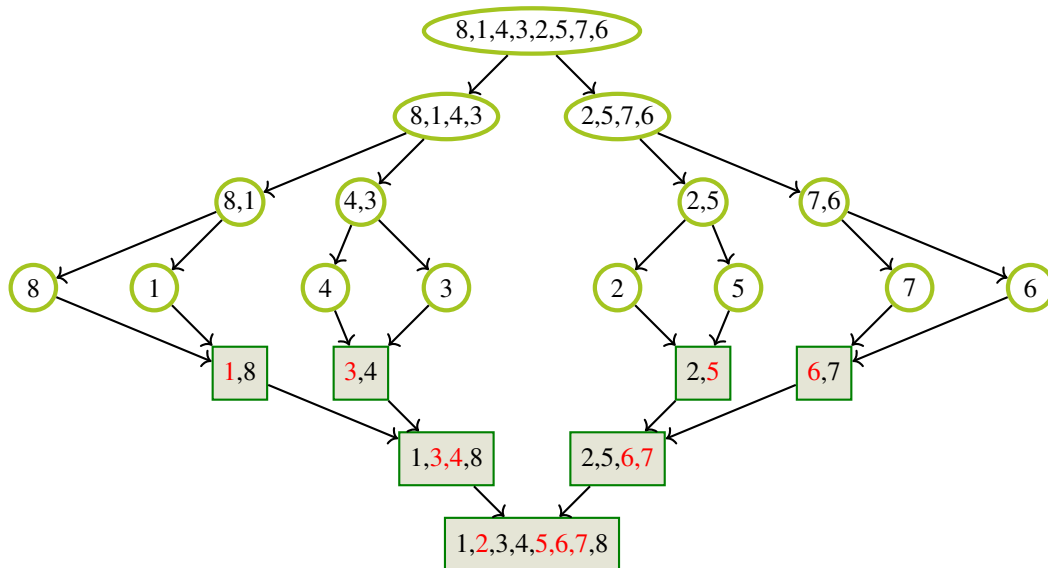


## 5.4 併合整列 (Merge Sort)

再帰的な考え方の応用の例として、merge sort という整列手法を取り上げる。この手法は、与えられた配列を半分に分割し、それぞれを整列する。そのうえで、整列された 2 つの配列を一つにマージ (併合) する。左右の半分の配列の整列も、同じ手順を適用する。整列対象の配列はどんどん小さく (要素数ほぼ半分に) なるので、最終的には、要素数 1 の配列になって、整列の必要がなくなる。

図は、8,1,4,3,2,5,7,6 という配列を整列する際の処理の流れを示している。上半分で分割し、下半分で整列が行われる。赤字は併合の際に右側から来た要素を示す。配列の要素数を  $N$  とすると、行が  $O(\log N)$  あり、各行あたりで必要な計算が  $O(N)$  なので、全体で  $O(N \log N)$  という計算量が導かれる。

プログラムとしては、分割と併合の 1 段のみ書けば十分なことに注意せよ。冒頭の `sum(n)` の例題で、`sum(n-1)` の処理を任せて `n` の加算のみ書いたように、分割された部分をソートするのは再帰先に任せることができる。



例題	併合	(金子)
整数を要素とし昇順に整列された配列 $a, b$ を与えられて、全ての要素を昇順に並べた新しい配列を返す関数 <code>merge(a, b)</code> を作成せよ。		

```
>>> merge([1, 3, 5], [2, 4]) 1
[1, 2, 3, 4, 5]                2
```

この併合の準備のために、回数を定めず条件が成り立たなくなるまで繰り返す、`while` という構文を用意する。

---

<b>while</b> 条件:	1
条件が成り立っている間繰り返し実行する文	2

---

Python3

---

```

1 def while_test(a):
2     while a > 0:
3         print("_", end="")
4         print(a)
5         a = a//2
6     return a

```

---



---

>>> while_test(128)	1
128	2
64	3
32	4
16	5
8	6
4	7
2	8
1	9
0	10

---



slice

Python では、配列  $a$  の区間  $[l, r)$  の部分の配列を取り出す記法として、 $a[l:r]$  という記法が用意されている。これを slice という。

Python3

---

```

1 def merge(a, b):
2     c = []
3     i, j = 0, 0
4     while i < len(a) and j < len(b):
5         if a[i] < b[j]:
6             c.append(a[i])
7             i += 1
8         else:
9             c.append(b[j])
10            j += 1
11        # ループ不変条件: a[0:i] と b[0:j] は昇順に c に詰まっている
12    c += a[i:] # a[i] 以降の a の要素を全て c に詰める
13    c += b[j:]
14    return c

```

---

例題	分割	(金子)
整数を要素とする配列 $a$ を与えられて、前半の配列を返す関数 <code>first_half(a)</code> と後半の配列を返す関数 <code>second_half(a)</code> を作成せよ。奇数の場合は前半の要素が一つ多いものとする。		

```

>>> a = [0,1,2,3,4]
[0, 1, 2, 3, 4]
>>> first_half(a)
[0, 1, 2]
>>> second_half(a)
[3, 4]
>>> b = [0,1]
[0, 1]
>>> first_half(b)
[0]
>>> second_half(b)
[1]
```

問題	併合整列 (mergesort)	(金子)
整数を要素とする配列 $a$ を与えられて、全ての要素を昇順に並べた新しい配列を返す関数 <code>mergesort(a)</code> を作成せよ。		

```

>>> mergesort([8,3,4,1,5,9,6,7,2])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## 5.5 再帰と反復の応用

5.1 節の問題は `for` ループで、5.2 節の問題は `while` ループを用いることで、再帰を使わずに書くことができる。この節の問題は、再帰または反復の適切な方もしくは両方で書いて見よう。(この節の問題は、他が終わってから取り組むこと)

問題	Recursion / Divide and Conquer - The Number of Inversions* ( <code>inversion_count</code> ) (AOJ)
配列内の要素のペアで、 $a[i] > a[j]$ ( $i < j$ ) のものを数えたい。例えば配列 $[3\ 1\ 2]$ の中には (3,1) と (3,2) の二つのペアの大小関係が逆転している。愚直に次のようなコードを書くと、要素数 $N$ の自乗に比例する時間がかかる ( $O(N^2)$ )。Merge sort の応用で、半分に分割しながら数	

(問題 続き)

えると,  $O(N \log N)$  で求めることができる.

そのような関数 `inversion_count(a)` を作成せよ.

参 考: [http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_D](http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D)

■制約を満たす最小値を求める 配列から値を探す状況以外にも, 二分法の考え方をを用いると綺麗に解ける問題もある.

問題

Search - Allocation\* (`min_sufficient_load`)

(AOJ)

様々な重さの荷物を, 並べられた順にトラックに積むことを考える. 荷物の重さは並べられた順に配列 `w` に格納してあるとする. 全てのトラックの最大積載量は同じとする. `k` 台のトラックにすべての荷物を順に積む切するためには, 各トラックの最大積載量は最低どれだけ必要か. 「トラックの最大積載量」の最小値を求める関数 `min_sufficient_load(w, k)` を作成せよ.

参 考: [http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_4\\_D](http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_4_D)

ヒント: トラックの最大積載量  $P$  として二分探索し, 全ての荷物を積める最小値を求める. 積みきれない最小値がほしいので, 「積みきれない最大値」が存在する範囲を  $[l, h)$  で表現し, 上限  $h$  を減らす条件を `ok` として表現する. 求める答えは  $h$  となる.

`ok` 関数は, たとえば以下のように作成できる:

- 現在のトラックに積んだ重さ (初期値 0) とトラック台数 (初期値 1) を表す変数を作成
- 各荷物について, 現在のトラックに積んで最大積載量を超えないならそのまま積む (現在のトラックに積んだ重さを増やす)/そうでなければ新しいトラックに積む (トラックの台数を増やして, 現在のトラックに積んだ重さをその荷物に設定)
- 最後に台数を  $K$  と比較

■三分法 : 2 分法では区間を 2 つに分けて一つを除外したが, 似た考え方として区間を 3 つに分けて 1 つを除外する方法を紹介する.

問題

極値 \* (`extremum`)

(金子)

二次式  $ax^2 + bx + c$  は  $a > 0$  とすると, 最小値を持つ. 最小値を含む区間  $[l, r]$  を与えられて, 再帰的に範囲を絞ることで最小値を求める関数 `extremum(a, b, c, l, r)` を作成せよ.

ヒント:  $m_1 = (2l + r)/3, m_2 = (l + 2r)/3$  とすると, 最小値が存在する区間を,  $[l, m_2]$  または



(問題 続き)

$[m_1, r]$  に絞ることができる.

■ミニマックス探索 : オセロのような二人ゼロ和ゲームを考える. 終端 (終局時の) 局面  $p$  の各プレイヤーの得点を,  $(p$  の自分の石の数  $-p$  の相手の石の数) とする. プレイヤを固定して,  $\text{score}(p) = \text{「}p \text{の黒石の数} - p \text{の白石の数」}$  とすると, 黒は  $\text{score}$  を最大化する, 白は  $\text{score}$  を最小化するように打つことが目標となる. 互いに最善を尽くすことを仮定すると, 手番  $t \in \{\text{黒}, \text{白}\}$  の (終局とは限らない) 局面  $p$  の, 両者最善を尽くした結果  $f(p, t)$  は以下のように再帰的に定義される. ただし  $\text{succ}(p, t)$  は, 局面  $p$  の次の (局面, 手番) の組の集合とする:

$$f(p, t) = \begin{cases} \text{score}(p) & p \text{ が終端} \\ \max_{(s, t') \in \text{succ}(p, t)} f(s, t') & p \text{ が終端でない, かつ } t \text{ は黒番} \\ \min_{(s, t') \in \text{succ}(p, t)} f(s, t') & p \text{ が終端でない, かつ } t \text{ は白番} \end{cases}$$

問題

オセロ ★★ (othello)

(金子)

4x4 のオセロについて, 黒と白が互いに最善を尽くした場合の結果を計算するプログラムを作成せよ.

プログラムを書く際に  $\text{succ}(p)$  を陽に求めるのは効率が良くないので, 局面  $p$  の空白に手番の石をおいてみて, 合法手であれば (石を返せたなら) それを  $s \in \text{succ}(p)$  として計算し, そうでなければ  $\max$  や  $\min$  の計算から除外すれば良い. オセロのルールでは, パスは合法手が一手もないときのみ可能である. また石を返せない場所に置くことはできない. この演習では, それらのルールを多少変更しても良いが, 変更は明記すること.

ヒント: 局面を, 黒 1, 白 -1, 空白 0 の二次元配列で表す. 局面  $p$  で手番  $t$  (1 または -1) がマス  $x, y$  に打った後の局面を返す関数  $\text{place}(p, t, x, y)$  を作成し, 十分テストすると良い. 局面  $p$  でマス  $x, y$  に打てない場合は, 配列以外の目印となる値 (たとえば False) を返すとする.

## 5.6 この章の提出課題

以下のいずれかで合格:

- Section 5.4 までの問題 (5.4 を含む, タイトル部分に ★ なし) を全て Python で解く (通常コース)  
5.1-5.3 の問題は, 変数の破壊的代入を用いずに (i.e., 定数変数のみ使用して) 解くこと.  
なお提出は求めないが, 末尾再帰への書き換えと, 「m 進数」も習得しておいてほしい
- “Koch Curve” と Section 5.5 の問題のなかから 2 題以上解く. (通常コースと発展コースの間)  
“Koch Curve” は, 変数の破壊的代入を用いずに (i.e., 定数変数のみ使用して) 解くこと
- 「オセロ」を解く (発展コース)

提出条件:

- 各関数に対応するテストケースが記述されていること (ユニットテストは、テンプレートの末尾の class にまとめて一箇所に記述する。)
- ループを含むコードは、ループ不変条件が明確にコメントで記述されていること
- 冗長なコードがないか確認し、標準的な書き方に整えられていること
- 指定した課題が、指定した関数名で作成されていること
- 提出ファイルは、1.A 節を参考にして、[jupyter から作成した](#) weekN- (学生証番号).ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中か完成品か, 授業中に既に OK をもらっていただければその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など, (途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む. (何も苦労がなかった場合を除き, 自作したことが分かるだけの十分なエピソードを記述すること - 授業中に教員の OK をもらった場合は簡潔で良い)

## 5.A 関数と実行状態の管理

「文が上から下の実行されることと再帰呼び出しの関係」について、補足する。

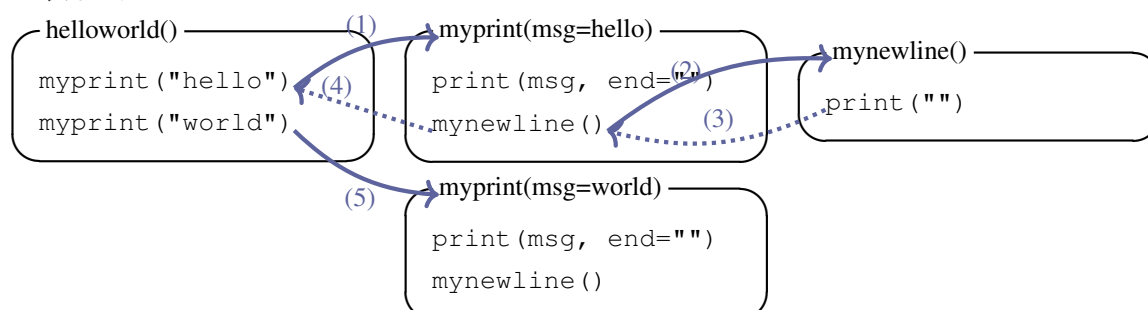
■通常の関数呼び出しの場合 多少作為的だが、次のようなソースコードがあったとする。関数 `helloworld` を実行すると、`hello` と `world` が一行ずつ表示される。

```

Python3 1 def mynewline():
2         print("")
3
4 def myprint(msg): # メッセージを書いて改行する
5         print(msg, end="")
6         mynewline()
7
8 def helloworld():
9     myprint("hello")
10    myprint("world")

```

この実行過程の一部を図で模式的に表すと以下のようになる。C++ を含む多くのプログラミング言語では、関数を呼ぶ (call) とその関数の実行状態や local 変数を管理する フレーム が作られる。下図ではフレームを枠囲みで表した。



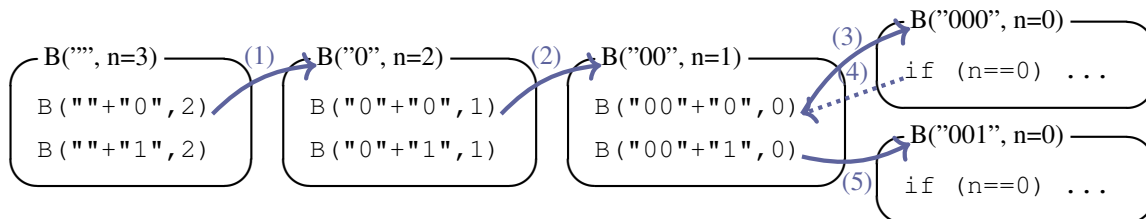
各フレーム内で見た時に、文は書かれた順に実行される。たとえば、一番左の `helloworld` のフレームにおいて `myprint("hello")` の実行が完了してから `myprint("world")` の実行が開始される。

`myprint("hello")` の実行が完了するとは、`myprint` 関数の呼び出しが実行され (矢印 (1))、そのフレーム内で、全ての文の実行を終えて制御が戻る (矢印 (4)) ことである。その過程では `mynewline` 関数も実行される (矢印 (2),(3))。矢印 (3) や (4) のように、あるフレームで全ての文の実行を終えるか `return` が起こると、呼び出し元の適切な位置に制御が戻される。

■再帰の場合 再帰の場合も同様に、呼出し毎にフレームが作られる。5.3 節の `m` 進数の例題を以下のように実装したとする。

```
Python3
1 def B(prefix, n):
2     if (n==0):
3         ...
4         return
5     B(prefix+"0", n-1)
6     B(prefix+"1", n-1)
```

この時、`B("", 3)` の呼出しは、以下のように進行する。



ソースコード上は同じ関数であっても、呼出し毎に異なるフレームが作成されることに注意する。それぞれのフレームごとに、引数、ローカル変数やどこまで実行したか、などの情報を維持する。このような情報の管理のために、スタック領域が用いられる (データ構造のスタックとは異なる)。実行が完了したフレームは破棄されるが、完了していないフレームはメモリ上に維持する必要がある。そのため、何段も関数を呼び出すと (たとえば 100 万)、実行時エラーになる場合がある。たとえば標準演習環境では、8 メガバイトが限度である。

```
ssh0-01m:~ 0123456789$ ulimit -a
...
stack size                (kbytes, -s) 8192
...
1
2
3
4
```

## 5.B ユークリッドの互除法と応用

例題

GCD and LCM

(AOJ)

正の整数 `a, b` に対する最大公約数、最小公倍数を求めるプログラムを作成せよ。

<http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0005&>

(例題 続き)

lang=jp

最大公約数は、ユークリッドの互除法 (参考書 [2, pp. 107–]) で求めると良い。

正の整数  $a \geq b$  に対する最大公約数を  $\gcd(a, b)$  と表記すると、

$$\gcd(a, b) = \begin{cases} a & (b = 0) \\ \gcd(b, a \% b) & (\text{otherwise}) \end{cases} \quad (5.2)$$

と定義できる。

問題

Extended Euclid Algorithm (`extend_euclid`)

(AOJ)

与えられた2つの整数  $a, b$  について  $ax + by = \gcd(a, b)$  の整数解  $(x, y)$  を求めよ。

[http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=NTL\\_1\\_E&lang=jp](http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=NTL_1_E&lang=jp)

拡張ユークリッド互除法で求められる。ユークリッド互除法が行っている計算で付随して求められ情報を活用する。

問題

中国剰余定理 (`chinese_remainder`)

(古典)

「3 で割ると 2 余り、5 で割ると 3 余り、7 で割ると 2 余る数はいくつか」一般に互いに素な整数  $a, b, c$  に対する剰余  $a', b', c'$  に対して、「 $a$  で割ると  $a'$  余り、 $b$  で割ると  $b'$  余り、 $c$  で割ると  $c'$  余る数」を一つ求める関数 `chinese_remainder` を作成せよ。

オンラインジャッジに適切な問題がなさそうなので、この問題は各自で入出力を作ってテストすること。テストしたデータをソースコードに添えること。

巨大な数を伴う計算を行う場合に、すべてを多倍長整数で計算すると計算コストが大きい。代わりに、複数の素数に対する剰余を計算して、最後に復元すると計算の効率が良い。囲碁 (19 路盤) の合法局面の数<sup>\*2</sup>は、このように求められた。

<sup>\*2</sup> 208168199381979984699478633344862770286522453884530548425639456820927419612738015378525648451698519643907259916015628128546089888314427129715319317557736620397247064840935 (DOI:10.1007/978-3-319-50935-8\_17)

## 第 6 章

# 木構造, 複合データ (クラス)

*Trees and classes*

2020-05-20 Wed

### 概要

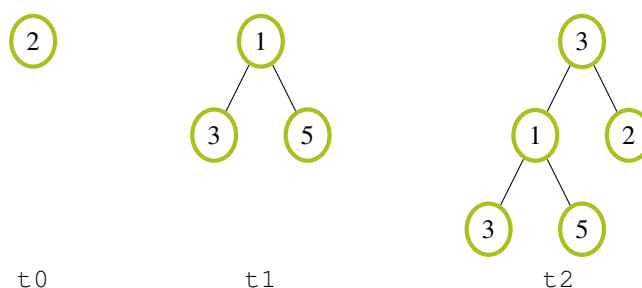
世の中のさまざまな概念は、階層的な構造を持っている。そのような概念をモデル化する技法の一つとして、木構造と、クラスを紹介する。演習時間の都合で同時に紹介するが、それぞれ独立の概念である。

## 6.1 階層的構造

データの中にデータがあり、その中にまた別のデータがある……というようなものを階層的なデータと言う。Python の配列は (C++ や C と異なり) 異なる型の要素を持つことができるので、配列を要素とする配列という形で木構造を書いてみよう。

### 6.1.1 数値の二分木

例として、数値の二分木を考える。二分木は空の木であるか、根という節点 (node) を持つ。各節点は数値を持ち、また左右の枝に二分木を持つ。二分木は空でも良いので、左右のどちらかあるいは両方の枝を持たない場合もある。



二分木に対する操作は、

- 構築 (make\_node),
- 根の値の参照 (value),

- 左右の部分木の参照 (left, right),
- 空の木かどうかの判定 (is\_empty)

とする. 空の木を `EmptyTree` と書く. 例として `make_node(2, EmptyTree, EmptyTree)` は図の `t0` 相当の木を作って返し, `left(t0)` は `EmptyTree` を返す. `make_node(1, make_leaf(3), make_leaf(5))` は図の `t1` 相当の木を作って返し, `left(t1)` は節点 3 を返す.

`make_node(3, make_node(1, make_leaf(3), make_leaf(5)), make_leaf(2))` は図の `t2` 相当の木を作って返し, `left(t2)` は `t1` 相当の木を返す.

ただし `make_leaf(x)` は, `make_node(x, EmptyTree, EmptyTree)` の略記である.

このような木を配列で表現するとすると, 次のように定義できる:

```

Python3 1 def make_node(num, left, right):
2     return [num, left, right]
3
4 def value(tree):
5     return tree[0]
6
7 def left(tree):
8     return tree[1]
9
10 def right(tree):
11     return tree[2]
12
13 # また, 特殊ケースとして空の節点を None で表わすことにする
14 EmptyTree = "EmptyTree" # None や [] など様々な候補がある, make_node の表現と
    重ならなければ良い
15 def is_empty(tree):
16     return tree == EmptyTree
17
18 # 便利のために, 子を持たない節点 (=葉) を作るための略記法を用意
19 def make_leaf(num):
20     return make_node(num, EmptyTree, EmptyTree)

```

動かしてみよう

```
>>> t0 = make_leaf(2)
>>> t0
[2, None, None]
>>> t1 = make_node(1, make_leaf(3), make_leaf(5))
>>> t1
[1, [3, None, None], [5, None, None]]
>>> t2 = make_node(3, t1, t0)
>>> t2
[3, [1, [3, None, None], [5, None, None]], [2, None, None]]
>>> value(right(left(t2)))
5
```

要素数 3 の配列: 要素はそれぞれ 2, None, None

要素数 3 の配列: 2 つの要素は, それ自身も配列

### 6.1.2 木の頂点の訪問

このようにして作成された数値の二分木について, 木の性質を調べる関数を作成しよう。

■節点の数 木  $t$  の節点の個数  $C(t)$  は, 左右の枝の個数の合計を考えて, 次のような式で表せる:

$$C(t) = \begin{cases} 0 & t \text{ が空のとき} \\ 1 + C(\text{left}(t)) + C(\text{right}(t)) & \text{それ以外} \end{cases}$$

例題	節点の個数	(金子)
木 <code>tree</code> の節点の個数を数える関数 <code>count_node(tree)</code> を, 再帰を用いて作成せよ。		

実行例:

```
>>> count_node(t0)
1
>>> count_node(t1)
3
>>> count_node(t2)
5
```

1  
2  
3  
4  
5  
6

回答例: これまでの再帰関数の実装と同様に, 式に対応させたコードを書けば良い。

```
Python3 1 def count_node(t):
2     if is_empty(t):
3         return 0
4     else:
5         return 1 + count_node(left(t)) + count_node(right(t))
```

## 例題

## 節点の数の合計

(金子)

木 `tree` の各節点が表す数を合計する関数 `sum_node(tree)` を, 再帰を用いて作成せよ。

木  $t$  の節点の数の合計  $S(t)$  は次のような式で表せる:

$$S(t) = \begin{cases} 0 & t \text{ が空のとき} \\ \text{value}(t) + S(\text{left}(t)) + S(\text{right}(t)) & \text{それ以外} \end{cases}$$

実行例:

```
>>> sum_node(t0) 1
2 2
>>> sum_node(t1) 3
9 4
>>> sum_node(t2) 5
14 6
```

## 問題

節点の持つ値の最大値 (`max_value`)

(金子)

木 `tree` の各節点が表す数の最大値を関数 `max_value(tree)` を, 再帰を用いて作成せよ。但し, `tree` の各節点が表す数は正と仮定して良い。

ヒント: まずは式を作ってみよう

実行例:

```
>>> max_value(t0) 1
2 2
>>> max_value(t1) 3
5 4
>>> max_value(t2) 5
5 6
```

## 例題

## 木の深さ

(金子)

木 `tree` の深さ返す関数 `depth(tree)` を, 再帰を用いて作成せよ。深さとは, 根から各節点までの距離 (たどる辺の数) の最大値とする。

ヒント: まずは式を作ってみよう



```

>>> depth(t0)      1
0                    2
>>> depth(t1)      3
1                    4
>>> depth(t2)      5
2                    6

```

■**preorder, inorder, postorder** 木を再帰的に訪問し節点を一度ずつ処理する順序として, preorder (自分, 左, 右), inorder (左, 自分, 右), postorder (左, 右, 自分) などが用いられる.

例題	行きがけ順	(金子)
木 <code>tree</code> の各節点の数値を, <code>preorder</code> の順に並べた配列を返す関数, <code>preorder(tree)</code> を作成せよ.		

`preorder` の順に並べた配列を  $P(t)$  とすると, 次のように定義される.

$$P(t) = \begin{cases} [] & t \text{ が空のとき} \\ [\text{value}(t)] + P(\text{left}(t)) + P(\text{right}(t)) & \text{それ以外} \end{cases}$$

ただし  $+$  は配列の結合を表すとする. 配列の結合方法については, 3.5 節で紹介した.

実行例:

```

>>> preorder(t0)    1
[2]                  2
>>> preorder(t1)    3
[1, 3, 5]            4
>>> preorder(t2)    5
[3, 1, 3, 5, 2]      6

```

問題	通りがけ順 ( <b>inorder</b> )	(金子)
木 <code>tree</code> の各節点の数値を, <code>inorder</code> の順に並べた配列を返す関数, <code>inorder(tree)</code> を作成せよ.		

実行例:

```

>>> inorder(t0)     1
[2]                  2
>>> inorder(t1)     3
[3, 1, 5]            4
>>> inorder(t2)     5

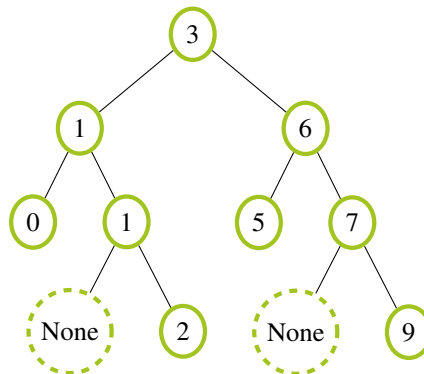
```

[3, 1, 5, 3, 2]

6

### 6.1.3 規則を持つ木: 二分探索木

数値の二分木の部分集合として以下の規則を満たす木を考える: 各節点の数値は, 左側の枝に現われるどの数値よりも大きく, 右側の枝に現われるどの数値よりも小さいまたは等しい. (これを 二分探索木条件 と呼ぶ).



例題

節点の追加

(金子)

二分探索木 `tree` と数値 `x` を与えられて, `x` の節点を `tree` に加えた新しい二分探索木を作成して返す関数 `add_node(tree, x)` を作成せよ.

注: 引数 `tree` として, 二分探索木条件を満たすもののみを対象とする. また元の `tree` を変更するのではなく, `tree` の構成要素を利用して良いので, 新たな木を作成せよ. ヒント:

)

木  $t$  に  $x$  を加えた二分探索木を  $B(t, x)$  とすると, 次のように定義可能である:

$$B(t, x) = \begin{cases} x \text{ を要素とする葉} & t \text{ が空のとき} \\ t \text{ の } \text{left}(t) \text{ を } B(\text{left}(t), x) \text{ で置き換えた木} & x < \text{value}(t) \\ t \text{ の } \text{right}(t) \text{ を } B(\text{right}(t), x) \text{ で置き換えた木} & \text{それ以外} \end{cases}$$

`make_leaf` や `make_node` を用いて, 再帰的に定義せよ.

問題

二分探索木の作成 (`make_binary_search_tree`)

(金子)

与えられた数値の配列 `a` の各要素を節点として持つ二分探索木を作成して返す関数 `make_binary_search_tree(a)` を作成せよ.

回答例: `EmptyTree` に順に要素を `add_node(tree, x)` する.

実行例

```

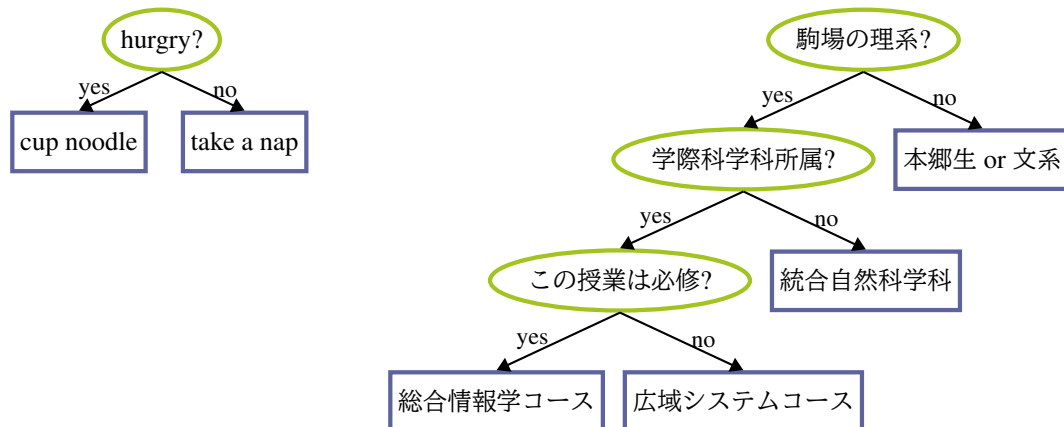
>>> t = make_binary_search_tree([3,1,6,1,7,9,2,0,5])
>>> t
[3, [1, [0, None, None], [1, None, [2, None, None]]],
     [6, [5, None, None], [7, None, [9, None, None]]]]
>>> count_node(t)
9
>>> inorder(t)
[0, 1, 1, 2, 3, 5, 6, 7, 9]

```

二分探索木の要素を通りがけ順 (inorder) に出力すると、昇順に並んでいることを確認する。

#### 6.1.4 決定木

「はい」か「いいえ」で答えるような質問によって枝分かれをして、最終的に適当なメッセージを表示するシステムを作ろう。



システムの状態は (1) 質問中 (図の楕円) か (2) 答え (図の長方形) のどちらかである。各状態に木の節点 (node) を対応させる。質問中での状態は、(a) 質問文の文字列 (b) 「はい」と答えた場合の次の状態 (c) 「いいえ」と答えた場合の次の状態とする。答えは文字列だけを持つ。

以下のような関数があれば、状態を管理できるだろう:

- 質問中での状態に関して、ノードを作る関数 `make_question_node(question, yes_branch, no_branch)`・質問文または左右の分岐を取り出す関数 `question_text(qnode)`, `question_yes(qnode)`, `question_no(qnode)`
- 答えを作成する関数 `make_answer_node(answer)`・答えの文字列を参照する関数 `answer_text(anode)`
- 状態が (1) か (2) のどちらかを判定する `is_question_node(node)`

システムは現在の状態に応じて適切な表示を行う。(1) の場合は、問題文と「yes」「no」の質問を表示する。キーボード入力に応じて選ばれた状態の表示を行う。(2) の場合は、メッセージを表示する。これらは、次のように定義できる:

```

Python3 1 # 状態に応じて適切な表示を行う

```

```

2 def display(node):
3     if is_question_node(node):
4         # 質問の場合, 質問文を表示して y/n をキーボードから読み込む
5         print("Q:_"+question_text(node)+ "_?_[yn]")
6         yn = input()
7         if yn.lower()[0] == "y":
8             display(question_yes(node))
9         else:
10            display(question_no(node))
11    else:
12        # 答えの場合, メッセージを表示する
13        print("---")
14        print("Answer:_"+ answer_text(node))

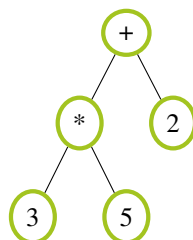
```

問題	Q&A	(金子)
<p>以下の必要な関数を定義し, 適当な質問・答えを作り, display に適用せよ。(関数群はテストケースを添える. 適当な質問・答えの実行結果はコピーペーストする)</p> <ul style="list-style-type: none"> <li>質問中の状態に関して, ノードを作る関数 <code>make_question_node(question, yes_branch, no_branch)</code>・質問文または左右の分岐を取り出す関数 <code>question_text(qnode), question_yes(qnode), question_no(qnode)</code></li> <li>答えは文字列だけを持つ. 作成関数 <code>make_answer_node(answer)</code>・答えを参照する関数 <code>answer_text(anode)</code></li> <li>状態が(1)か(2)のどちらかを判定するための関数, <code>is_question_node(node)</code></li> </ul>		

ヒント: 数値の二分木の定義を参考にせよ

### 6.1.5 四則演算の構文木

また別の種類の木として, 内部節点に四則演算を表す文字, 葉に整数を持つ二分木を考える. 木は値を持ち, 葉の値は所持する整数, 内部節点の値は左右の木の値をその節点の演算で計算したものとする. なお, 節点は子を持たないか(=葉), 2つの子を持つとする. 以下の例で, 根の値は  $((3 \times 5) + 2) = 17$  である.



問題

四則演算 (calc)

(金子)

四則演算を表す構文木 `tree` を与えられて、根の計算結果を返す関数 `calc(tree)` を作成せよ。

```
>>> t = make_node("+", make_node("*", make_leaf(3),
1
    make_leaf(5)), make_leaf(2))
2
>>> t
3
["+ ", ["*", [3, None, None], [5, None, None]], [2, None, None]]
4
>>> calc(t)
5
17
6
```

ヒント: 節点 `t` が子を持つかどうかは, `is_empty(left(t))` など判定できる。

### 6.1.6 内部表現の世界と使用する世界の分離

これまで木の内部表現として配列を用いてきた。別案として、これを `dict`(連想配列) を用いて表現することもできる (詳しくは 9.4 節で取り扱う)。6.1.1 節で定義した `make_node`, `value`, `left`, `right` の 4 つの関数の実装を以下のように取り替えてみよう。残りの関数はそのままでも、二分探索木などで作成したプログラムをそのまま動かすことができる。

Python3

```
1 def make_node(num, left, right):
2     return {"number": num, "left": left, "right": right}
3
4 def value(tree):
5     return tree["number"]
6
7 def left(tree):
8     return tree["left"]
9
10 def right(tree):
11     return tree["right"]
```

ここでは連想配列のキーは、文字列で表現した。他には整数やタプルでも表現できる。

多くの場合、表現したい内容に対して (今回は二分木)、それを表現する方法は複数通りある。内部表現次第で性能に影響する場合もあるので、注意深く選ぶ必要がある (今回の配列と `dict` では大差ない)。また、内部表現をあとから取り替える場合は、書き換え箇所が少なければ少ないほど望ましい。今回の `make_node`, `value`, `left`, `right` の 4 つの関数のように、適切に抽象化を行うことは、書き換え箇所を抑えることにつながる。なお、Q&A のプログラムで、内部状態を変更するには同様に、`make_question_node`, `question_text`, `question_yes`, `question_no`, `make_answer_node`, `answer_text`, `is_question_node` を書き換える。

## 6.2 クラスの利用の初歩

関連の深いデータをまとめる方法として、クラス (の機能の一部) を導入する。保健センターなどの業務で、複数人の身長と体重に興味があるとする。身長も体重も数であるので、それぞれは Python で表現可能であるが、それらをまとめたデータを作りたい。天下り式の説明であるが以下のようにクラスを用いると 2 つの変数 height と weight がセットになったデータ型を作ることができる。

```
Python3 1 class Person:
2     def __init__(self, height, weight):
3         self.height = height
4         self.weight = weight
```

1 行目 class から 4 行目までのブロックが、Person と名付けられたクラスの定義である。2 行目から 4 行目のような、関数と似た記述はメソッドと呼ぶ (たとえばアルゴリズム入門の教科書 [1] 8 章を参照)。特に\_\_init\_\_という名を持つ特別なメソッドは、初期化のために用いられる。この意味も追って説明する。使いかたは次のようになる。

```
>>> taro = Person(180,90) 1
>>> taro 2
<__main__.Person object at 0x10542b9e8> 3
>>> taro.height 4
180 5
>>> taro.weight 6
90 7
>>> jiro = Person(160,55) 8
>>> jiro 9
<__main__.Person object at 0x10542ba90> 10
>>> jiro.height 11
160 12
```

Python で決められた構文である Person(height, weight) を呼ぶたびに、新しいデータの組が作られる (インスタンスと呼ぶ)。それらは height と weight で参照可能なデータの組を持つ。名称が同じ height であっても、インスタンスが異なれば、すなわち taro と jiro では異なる値を持つ。それぞれが持つ値は Person(180, 90) のように Person に与えた引数に対応し、\_\_init\_\_という特別な名前を持つメソッドにより内部変数 height と weight にコピーされる。

クラスに、メソッドを追加することで様々な機能を実現できる。たとえば bmi を計算する機能を付与してみよう。

```
Python3 1 class Person:
2     # 以下を追加
3     def bmi(self):
4         return self.weight / (self.height/100.0)**2
```

実行例:

```
>>> taro.bmi() 1
27.77777777777775 2
>>> jiro.bmi() 3
21.484374999999996 4
```

ここまでの資料では二分木を, 内部表現として配列あるいは dict を使って実現した. この内部表現にクラスを用いて実現してみよう.

## 問題

## クラスを用いた木 (Node)

(金子)

木の節点を表すクラス Node を適切に定義せよ.

動作確認として, 6.1.1 節で定義した木の機能を次のように変更しても, 二分探索木または四則演算が Node を対象に動作することを確認せよ.

Python3

```
1 def make_node(num, left, right):
2     return Node(num, left, right)
3
4 def value(tree):
5     return tree.value
6
7 def left(tree):
8     return tree.left
9
10 def right(tree):
11     return tree.right
```

注: EmptyTree や is\_empty の定義は変更不要である. また上記は (メソッドではなく) 通常の間数である.

## 6.3 発展問題

## 問題

## Reconstruction of a Tree 改題 \* (recover)

(AOJ 改題)

ある二分木について異なる頂点には異なるラベルがついているとする. preorder で出力した頂点のリストと, inorder で出力したリストを与えられて, 元の木を復元して返す関数 recover(preorder, inorder) を作成せよ.

(\* 付き問題にもテストケースも添えること.)

原 題: <http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=>

	(問題 続き)
ALDS1_7_D	

ヒント: 部分木の頂点を求めて分割統治しよう

問題	構文解析 ★ (parse)	(金子)
<p>四則演算を表す文字列 <math>s</math> に対応する木を返す関数 <code>parse(s)</code> を作成せよ。</p> <p>簡単のため文字列 <math>s</math> は, 1 以上 100 未満の整数と四則演算と括弧のみを用いた演算を表すとする。また演算子を用いる際には必ず括弧が用いられるとする。すなわち "<math>1+2+3</math>" は与えられない。"<math>((1+2)+3)</math>" または "<math>(1+(2+3))</math>" が与えられる。</p>		

問題	二分探索木 (親の管理)★ (successor)	(金子)
<p>「クラスを用いた木」について, <code>Node</code> の親を管理するフィールド <code>self.parent</code> を追加し, 親を得る関数 <code>parent(tree)</code> を定義せよ。また <code>make_node</code> を適切に変更せよ。そのうえで, ある二分探索木 <code>tree</code> の任意の節点 <code>node</code> について, 「次」の節点を返す <code>successor(node)</code> を定義せよ。「次」とは, 二分探索木の節点を通りがけ順 (inorder) に並べた際の順序で考える。次の節点が存在しない場合は <code>EmptyTree</code> を返すとする。</p> <p>ヒント: 右の節点を持つかどうかで場合分けして考えよう。</p> <p>(オプション: 二分探索木 <code>tree</code> の任意の節点 <code>node</code> について, 節点 <code>node</code> を削除した木を作る関数 <code>delete(tree, node)</code> を作成せよ。削除後の木も二分探索木条件を満たすようにすること)</p>		

問題	Q&A-C++★	(金子)
<p>「Q&amp;A」または「二分探索木 (親の管理)」を C++ (C++11) で作成せよ。</p> <p>節点を表す <code>Node</code> クラスを作成し, 左右の枝としての c++11 の <code>shared_ptr&lt;Node&gt;(Node 型のスマートポインタ)</code> を持つものとする。<code>shared_ptr&lt;Node&gt;</code> については, 8 章 (8.5.1 節) も参照。</p>		

## 6.4 この章の提出課題

以下のいずれかで合格:



- 問題 (タイトル部分に★なし) を全て Python で解く (通常コース)  
ただし, 「節点の持つ値の最大値」は提出不要
- 星付きの問題 2 題以上解く (発展コース)  
ただし, 一題は「クラスを用いた木」でも良い

提出条件:

- 各関数に対応するテストケースが記述されていること (ユニットテストは, テンプレートの末尾の class にまとめて一箇所に記述する.)
- ループを含むコードは, ループ不変条件が明確にコメントで記述されていること
- 冗長なコードがないか確認し, 標準的な書き方に整えられていること
- 指定した課題が, 指定した関数名で作成されていること
- 提出ファイルは, 1.A 節を参考にして, [jupyter から作成した](#) weekN- (学生証番号).ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中か完成品か, 授業中に既に OK をもらっていればその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など, (途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む. (何も苦労がなかった場合を除き, 自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

## 第 7 章

# 状態管理，抽象インターフェースと継承

*internal state, abstract interface and inheritance*  
2020-05-27 Wed

### 概要

先週は，value, left と right のような組となるデータをまとめて扱うための道具としてクラスとオブジェクトの概念を導入した．今週は，オブジェクトの内部状態の変化も含めて，再度クラスとオブジェクトについて演習する．インターフェースを限定することで，複雑なプログラムを管理しやすくする点が，先週から続く主題である．

身に付ける目標は，クラスとオブジェクト (インスタンス)，継承と合成，関数にオブジェクトを渡す際のインターフェースの考え方などである．

## 7.1 内部状態を持つオブジェクト

### 7.1.1 内部状態

オブジェクトの集合によって実世界をモデル化する場合には，個々のオブジェクトは identity を保ちながら，内部状態が変化してゆくと考ええる．例えば，銀行口座をオブジェクトとする場合，同じ口座という identity を保ったまま残高が変化してゆく．

#### 例題

#### 銀行口座

(原題増原, 金子が翻案)

銀行口座を表すクラス BankAccount を作成せよ．口座作成時の金額 (たとえば 1 万円) を与えて BankAccount (10000) と口座に相当するオブジェクトを作成する．そのように作成したオブジェクトに対してメソッド withdraw(x) を呼ぶと口座から x 円引き出して，残高を返すものとする．もし残高よりも沢山引き出そうとした場合は “insufficient funds” というメッセージを返す．

はじめにプロトタイプとして，以下の実装例の BankAccount0 について考えてみる．(あとで，改良した BankAccount を導入する)

Python3

```

1 class BankAccount0:
2     def __init__(self, initial_balance):
3         self.balance = initial_balance
4
5     def withdraw(self, amount):
6         if self.balance >= amount:
7             self.balance -= amount
8             return self.balance
9         else:
10            return "Insufficient_funds"

```

1 行目 class から 10 行目までのブロックが、BankAccount と名付けられたクラスの定義である。2 行目から 3 行目、5 行目から 10 行目では、それぞれ\_\_init\_\_と withdraw という二つのメソッドを定義する。

6.2 節で紹介したように、インスタンス作成時に、BankAccount0 で与えた変数で、\_\_init\_\_が呼ばれ、ここでは変数 self.balance が初期設定される。

このクラスを用いて、BankAccount (初期残高) が実行される度に新しく口座ができる。BankAccount で新しいインスタンス (オブジェクト) が作成され、その度に、\_\_init\_\_が一度呼ばれる。その引数は、BankAccount に与えられた引数 (上記の例では 1 万) である。実行例 3 行目の withdraw のように、メソッドを実行する際には、インスタンス (この例では a や b) にドットで続けて記述する。メソッド内で使われている self.balance はインスタンス変数と呼ばれる特殊な変数で、BankAccount(初期残高) して作られたインスタンス毎に (この例では a や b では) 異なる値を持つ。引数 self が、操作しているインスタンス a や b を指す。

使用例:

```

>>> a = BankAccount0(10000) # aさんの口座を一つつくる      1
>>> b = BankAccount0(10000) # bさんの口座を一つつくる      2
>>> a.withdraw(5000)                                          3
5000                                                           4
>>> b.withdraw(7000)                                          5
3000                                                           6
>>> b.withdraw(4000)                                          7
"Insufficient_funds"                                           8
>>> a.withdraw(4000)                                          9
1000                                                            10

```

この BankAccount0 クラスについて、withdraw メソッドが意図通りに動作することを確認する。

## 7.1.2 カプセル化

銀行口座の例の balance のように、インスタンス変数の一部は、非公開であることが望ましい場合がある。非公開とは、対応するメソッド (withdraw など) のみから変数を操作可能であること、つまり (特に裏口を作らなければ) 外部から直接操作することはできないことを指す。内部状態を外部から触れなくすることを一般に「カプセル化 (encapsulation)」と言う。

```
>>> a.balance += 10000000000 # 違法ハッキング (?) で財産拡大      1
(成功してしまった)                                           2
```

現状の BankAccount0 は、このようなコードも警告なしに実行してしまう。すなわちカプセル化は行われていない。Python では、プログラムの設計者が、カプセル化の意図を使用者に伝える方法として、2つのアンダースコア `__` をつける方法がある。<sup>\*1</sup>

```
Python3 1 class BankAccount:
2     def __init__(self, initial_balance):
3         self.__balance = initial_balance
4
5     def withdraw(self, amount):
6         if self.__balance >= amount:
7             self.__balance -= amount
8             return self.__balance
9         else:
10            return "Insufficient_funds"
```

このようにすると、先程の操作がエラーになることを確認せよ。

```
>>> a.__balance += 10000000000      1
(エラー)                             2
```

これは Python が2つのアンダースコア `__` を別の名前に付け替えるためである。この文法は厳密なアクセス制限とは異なり、実際には `a._BankAccount__balance += 10000` とすると成功する。しかし、意図しない変数の破壊を防ぐという意味では十分であるため、本演習では、今後カプセル化する変数は、名前に2つのアンダースコア `__` をつける方針を採用する。なお、アンダースコア1つのみを用いる方法を推奨する主張もある。<sup>\*2</sup>

引き出ししかできない口座だと、残高は減る一方なので、預金もできるようにする。そのためには以下のよう `deposit(amount)` メソッドを定義する。

```
Python3 1 class BankAccount:
2     # 先ほどのクラス定義に追加
3     def deposit(self, amount):
4         self.__balance += amount
```

使用例

```
Python3 1 a = BankAccount(10000)
2     # <_main_.BankAccount object at 0x10542be80>
3     a.withdraw(5000)
```

<sup>\*1</sup> (double underscores を略して dunder と呼ばれる)

<sup>\*2</sup> <https://www.python.org/dev/peps/pep-0008/>

```

4 # 5000
5 a.withdraw(6000)
6 # "Insufficient funds"
7 a.deposit(4000)
8 # 9000
9 a.withdraw(6000)
10 # 3000

```

## 例題

## 暗証番号付き口座

(増原)

クラス BankAccount に、暗証番号を付け加えてみよ。

- initialize において、残高の初期値に加えて、暗証番号に相当する文字列を受け取って新しい口座を作る
- さらに、withdraw, deposit の手続きには、引き出し/預金額に加えて暗証番号を受け取り、口座を作ったときと同じ暗証番号であった場合のみ操作ができるようにする
- 暗証番号が間違っていた場合は、“Incorrect password” というメッセージを返すものとする

実行例:

Python3

```

1 acc = BankAccount2(10000, "secret")
2 # <_main_.BankAccount2 object at 0x10542bef0>
3 acc.withdraw(4000, "secret")
4 # 6000
5 acc.deposit(5000, "some-other-password")
6 # "Incorrect password"

```

## 例題

## カウンター

(金子)

次のような数を数える機能を持つクラス Counter を作成せよ。Counter によって作成される。初期値は0とする。メソッド increment で数を1増やす。メソッド counter で現在の数を返す。

実行例

```

>>> c = Counter()
>>> c.increment()
1
>>> c.count()
1
>>> c.increment()
2

```

1  
2  
3  
4  
5  
6  
7

```
>>> c.count()
2
```

実装例:

```
Python3 1 class Counter:
2     def __init__(self):
3         self.__counter = 0
4
5     def increment(self):
6         self.__counter += 1
7
8     def count(self):
9         return self.__counter
```

まず、数を数える変数をカプセル化する、と判断したとする。つまりカウンターの値を変更するには、`increment` メソッドを呼ぶ必要がある、という設計を採用した。そこで変数の名前を、現在の数を返すメソッド `count` に合わせて、`__counter` とする。new の際に `__init__` が呼ばれるというルールを思い出し、そこで初期値に初期化する。続いて、メソッド `increment` と `count` を定義する。

#### 問題

#### 時計 (ClockTimer)

(増原)

時計の時を刻むクラス `ClockTimer` を定義せよ。このクラスは、次のようなインターフェイスを持っている:

- `ClockTimer(hour, minute, second)` によって、hour 時 minute 分 second 秒を表わす時計が作られる。
- インスタンス変数 (あるいはプロパティ) `hour, minute, second` によって、その時計が表わしている時刻が得られる。
- メソッド `tick` によって 1 秒時刻が進む。

```
>>> timer = ClockTimer(22, 54, 14)
>>> timer.hour
22
>>> timer.minute
54
>>> timer.second
14
>>> timer.tick()
15
>>> timer.hour
22
>>> timer.minute
```

54	13
>>> timer.second	14
15	15
>>> for _ in range(45): # 45 回実行	16
...     timer.tick()	17
...	18
>>> timer.hour	19
22	20
>>> timer.minute	21
55	22
>>> timer.second	23
0	24

今回は、時刻を表す状態変数 hour, minute, second について、カプセル化しないと判断したとする。すなわち、時刻を変更する方法としては、メソッド tick を呼ぶことと (時が 1 秒進んだ状況に相当)、変数に直接代入する (時計の針を無理やり動かした状況に相当) 方法の、両方を想定する。

### 7.1.3 メソッドの再帰呼出し

クラスでモデル化するものは、必ずしも実世界のモノに対応する必要はない。ここでは、(恣意的だが) フィボナッチ数を計算する計算主体というモデルを考えてみよう。

Python3

```
1 class Fib:
2     def fib(self, n):
3         if n == 0:
4             return 0
5         elif n == 1:
6             return 1
7         else:
8             return self.fib(n-1)+self.fib(n-2)
```

>>> f = Fib()	1
>>> f.fib(3)	2
2	3
>>> f.fib(4)	4
3	5
>>> f.fib(10)	6
55	7
>>> f.fib(20)	8
6765	9

## 例題

## 呼び出し回数測定

(金子)

上記の Fib クラスを拡張して, メソッド fib が何回呼ばれたかを測定する機能をもたせよ. またメソッド count によってその回数分かるようにせよ.

道具として Counter クラスを組み合わせると簡単に実現することができる. このようにあるクラスの部品として別のクラスを用いることを composition と言う.

Python3

```

1 class Fib:
2     def __init__(self):
3         self.__counter = Counter()
4
5     def fib(self, n):
6         self.__counter.increment()
7         if n == 0:
8             return 0
9         elif n == 1:
10            return 1
11        else:
12            return self.fib(n-1)+self.fib(n-2)
13
14    def count(self):
15        return self.__counter.count()
```

使用例:

```

>>> g = Fib()
>>> g.fib(3)
2
>>> g.count()
5
>>> g.fib(10)
55
>>> g.count()
182
>>> h = Fib()
>>> h.count()
0
```

## 問題

## メモ化 (MemoFib)

(金子)

これまでのフィボナッチ数の計算はメソッド fib(n) が同じ n に対して何度も呼ばれるため, 計算効率が悪い. 改善のための一つの方法は, 配列 memo を用意し, fib(n) を計算するたびに



(問題 続き)

memo[n] に答えを記憶し, 次に fib(n) を計算する際に memo[n] に答えが既にあるば (あらためて fib(n) を計算する代わりに) memo[n] の値を使うという方法である. そのように計算を行うクラス MemoFib を作成せよ. なお簡単のため n は 1000 未満と仮定して良い.

このような工夫を一般に **メモ化** (memoization) と呼ぶ.

実行例 (memo の参照はカウントしないとする):

```
>>> m.fib(10) 1
55 2
>>> m.count() 3
11 4
>>> m.fib(20) 5
6765 6
>>> m.count() 7
21 8
>>> m.fib(20) 9
6765 10
>>> m.count() 11
21 12
```

実装例 (主要部分):

```
Python3 1 def fib(self, n):
2     if # self.__memo[n] に答えが入っていなかったら
3         self.counter.increment()
4         # 答えを計算して self.__memo[n] に代入
5     return self.__memo[n]
```

self.memo[n] は, self.fib(n) の計算結果と区別できる値で初期化しておく.

#### 7.1.4 状態と複製

クラスのオブジェクト, たとえば銀行口座を表す変数は「参照」である (何回か前の演習の 4.4 節などを参照のこと). したがって, b=a のように複製しても口座自体が複製されるわけではなく, 口座にアクセスするカードや通帳に相当する概念がコピーされるにすぎない. したがって以下のように, 複製された b を通じて引き落とせば, 元の a にも影響が及ぶ.

```
>>> a = BankAccount(10000) 1
>>> a 2
<__main__.BankAccount object at 0x103ec8588> 3
>>> a.withdraw(3000) 4
7000 5
```

```

>>> b = a
>>> b
<__main__.BankAccount object at 0x103ec8588>
>>> b.withdraw(7000)
0
>>> a.withdraw(1)
'Insufficient funds'

```

例題

Copy

(金子)

銀行口座の例について、口座ごと複製するメソッド `clone` を実装せよ。残高 5000 円の口座を `clone` すると新たに残高 5000 円の口座ができて、それぞれから 5000 円ずつ下ろせるものとする。

使用例:

```

>>> a = BankAccount(10000)
>>> a
<__main__.BankAccount object at 0x103ec8588>
>>> b = a.clone()
>>> b
<__main__.BankAccount object at 0x103ec85c0>
>>> b.withdraw(3000)
7000
>>> a.withdraw(9000)
1000

```

実装例:

Python3

```

1 class BankAccount:
2     def clone(self):
3         return BankAccount(self.__balance)

```

自分と同じ状態 (`self.__balance`) を持ったオブジェクトを新しく生成すれば良い。

問題

変更可能な二分探索木 (BSTree) (optional)

(金子)

先週作成した二分木のクラスを参考に、変更可能な二分探索木のクラス `BSTree` を用意せよ。さらに、これらを用いて、自分自身が二分探索木であるときに値を追加するメソッド `add(x)` を作成せよ。

空の木である `EmptyTree` や `is.empty` の定義は適当に変更して良い。

(問題 続き)

オプション: 二分探索木 `tree` の任意の節点 `node` について, 節点 `node` を削除した木を作る関数 `tree.delete(node)` を作成せよ. 削除後の木も二分探索木条件を満たすようにすること. `node` の子供の数が 0 または 1 の時は, `node` の代わりにその子を上につなげれば良い. `node` が子を 2 つ持つ時は, `node` の `successor` を元の木から取り除いて, `node` と差し替える. `successor` は性質上, 子をたかだか一つ持つ.

注: 先週作成した `add.node(tree, x)` は元の木 `tree` を一切変更せずに新しい木を作成した. 今回のような変更を行う場合は影響範囲を慎重に把握する必要がある. 次の問題も参照.

使用例: 以下の例では `self.value=None`, `self.left=None`, `self.right=None` であるような節点を `EmptyTree` と定義しなおしている.

```
Python3
1 >>> a = BSTree(None, None, None)
2 >>> a.add(3)
3 >>> for x in [1, 6, 1, 7, 9, 2, 0, 5]:
4 ...     a.add(x)
5 ...
6 >>> a.inorder()
7 [0, 1, 1, 2, 3, 5, 6, 7, 9]
```

問題

Deep Copy★

(金子)

前問で作成した木のクラスに, 木全体のコピーを行うメソッド `clone` を (`copy.deepcopy` を使わずに) 用意せよ.

`clone` 元の木にどのような変更 (左や右の部分木に対する変更も含む) を行っても, `clone` 後の木が変化しないように作成すること.

(★がついているようにそれほど簡単ではない. テストの方法も含めて検討せよ)

### 7.1.5 クラスの図表現

ソフトウェアを作る際に, 設計を議論する段階では, 具体的なソースコードで表現するよりも抽象化した図を用いる方が見通しが良いことが多い. クラスの図表現として広く使われている記法では, 四角でクラスを表す. そして, 3つに区切った領域にそれぞれ, クラス名, そのクラスのオブジェクトが持つ属性 (インスタンス変数), そのクラスのオブジェクトで行える操作 (メソッド) を記入する.

BankAccount
balance password
withdraw(amount, password) deposit(amount, password)

詳しくはUML や [クラス図](#) などのキーワードで調査してみよう。

## 7.2 インターフェース

オセロやすごろくなどのゲームをプレイするプレイヤを考える。複数のプレイヤにはそれぞれ個性があり、同じ状況でも異なる選択肢を選択するが、同じゲームをプレイするとする。このような場合にはゲームをプレイするのに必要なプレイヤの機能 (の外界とのやりとりの部分) を [インターフェース](#) として抽出し定義すると、見通し良く設計することができる。

### 7.2.1 囚人のジレンマゲーム

ここでは囚人のジレンマゲームを例として考える。このゲームには二人のプレイヤが登場し、各プレイヤは協調か裏切りのどちらかを選択し、ゲームマスターに提出する。ゲームマスターはその二人の選択の組み合わせに従って決まる報酬をそれぞれに与える。

	協調	裏切り
協調	2, 2	0, 3
裏切り	3, 0	1, 1

両者が協調すると、両者とも報酬があり両者の報酬の和が最大となる。しかし片方が裏切ると、裏切った方はすべての可能性の中で最大の報酬を得るが、裏切られた方の報酬はない。両者が裏切った場合は、一方的に裏切られるよりはましだが、両者の報酬の和は4通りの組み合わせの中で最低となる。

### 7.2.2 ゲームとプレイヤのモデル化

まず、プレイヤの行動の選択肢と報酬を定義する。プレイヤの行動は協調か裏切りの2種類なので、ここでは整数0と1で表すとして、CooperateとDefectと名づけた定数を設ける。それらに対応するように、二人のとした選択肢から報酬を返す `Reward[自分の選択][相手の選択]` という二次元配列を定義する。また、ゲームのルールに則った行動かどうかを検証する `valid_action` という関数を用意しておく。

```

Python3
1 Cooperate = 0
2 Defect = 1
3 def valid_action(act):
4     return act == Cooperate or act == Defect
5
6 Reward = [
7     [ 2, 0 ], # 自分が協調した
8     [ 3, 1 ], # 自分が裏切った

```

9 ]

プレイヤーにはどのような機能があれば良いだろうか? まずは, 協調か裏切りかを返すメソッド `play` が必須である. さらに結果を集計する便宜を考えて, 名前を返すメソッド `name` を持たせる.

例として次の3種類のプレイヤーを考える: 必ず協調する `CooperatePlayer`, 必ず裏切る `DefectPlayer`, ランダムに決める `RandomPlayer` である. どのクラスも `name` と `play` というメソッドを持っていることに注意する.

CooperatePlayer	DefectPlayer	RandomPlayer
name() play()	name() play()	name() play()

```

Python3 1 import random
2 class CooperatePlayer:
3     def name(self):
4         return "CooperatePlayer"
5
6     def play(self):
7         return Cooperate
8
9 class DefectPlayer:
10    def name(self):
11        return "DefectPlayer"
12
13    def play(self):
14        return Defect
15
16 class RandomPlayer:
17    def name(self):
18        return "RandomPlayer"
19
20    def play(self):
21        return Cooperate if (random.randrange(2) == 0) else Defect

```

続いてこれらのプレイヤー同士を戦わせる関数 `play_one_game` を定義する. 引数の `player_a` と `player_b` は, どちらも, 先に定義した `CooperatePlayer`, `DefectPlayer`, `RandomPlayer` のどれかであると仮定する.

```

Python3 1 def play_one_game(player_a, player_b):
2     act_a = player_a.play()
3     act_b = player_b.play()
4     if not valid_action(act_a): # 行動がルールに従っているかを確認
5         raise ValueError
6     if not valid_action(act_b): # 行動がルールに従っているかを確認
7         raise ValueError
8     reward_a = Reward[act_a][act_b]

```

```

9      reward_b = Reward[act_b][act_a]
10
11      print(player_a.name(), "_v.s._", player_b.name()) # 結果を表示
12      print("_actions:_", act_a, act_b)
13      print("_rewards:_", reward_a, reward_b)
14
15      return act_a, reward_a, act_b, reward_b

```

2,3 行目で両プレイヤーの行動を受け取り, 6,7 行目でそれぞれの報酬に変換し, 9-11 行目で結果を表示し, 13 行目の報酬のペアを関数の結果とする. ここで 6,7 行目は例外的なケースの確認を行うもので, プレイヤーがルール通りでない行動をした場合 (つまり協調でも裏切りでもない行動を選んだ) に, 「例外」として実行を停止する<sup>\*3</sup>. `raise` は Python で例外を送出するための言語機能である.

実行例は以下ようになる. `CooperatePlayer` と `DefectPlayer` は毎回同じ行動をとるが, `RandomPlayer` の行動は異なる場合がある.

Python3

```

1 >>> pc = CooperatePlayer()
2 >>> pd = DefectPlayer()
3 >>> pr = RandomPlayer()
4
5 >>> play_one_game(pc, pd)
6 CooperatePlayer v.s. DefectPlayer
7   actions: 0 1
8   rewards: 0 3
9   (0, 0, 1, 3)
10 >>> play_one_game(pc, pr)
11 CooperatePlayer v.s. RandomPlayer
12   actions: 0 0
13   rewards: 2 2
14   (0, 2, 0, 2)
15 >>> play_one_game(pd, pr)
16 DefectPlayer v.s. RandomPlayer
17   actions: 1 1
18   rewards: 1 1
19   (1, 1, 1, 1)

```

関数 `play_one_game` に 3 種類のどのプレイヤーを渡しても動くことに注目されたい. 仮にもし, 対戦するプレイヤーの組み合わせごとに `play_one_game_random_versus_cooperate` のように別々の関数を作る必要があるとしたら, 耐えられない手間となってしまう. この例では, 3 種類のプレイヤーがどれも共通に `name` と `play` という一字一句同じ名前のメソッドを持っている点が重要である. 外から見て (使う立場で) オブジェクトがどのような振る舞いをするか (≒ どの名前のメソッドを呼べるか) を, インターフェースと呼ぶ. 3 種類のプレイヤーは共通のインターフェースを持っている. 特定のインターフェースをクラスに持たせる方法は, プログラミング言語によって異なる.

<sup>\*3</sup> 複数の作者のプレイヤーが競うような状況では, もしかするとバグを期待してルールの穴をつくプレイヤーがいるかもしれない? 演習では, 何かを大きく間違えた時に早く気づくことができるというメリットがある.

### 7.2.3 繰り返し囚人のジレンマゲーム

ゲームを少し拡張して繰り返し囚人のジレンマゲームを作ってみよう。このゲームでは、同じプレイヤー同士で、囚人のジレンマゲームを繰り返し行う。もし相手が常に裏切るプレイヤーならば協調するメリットはないが、協調も視野にいたれた相手であればこちらも協調を選択肢に含めるほうが (1,1) よりも (2,2) の報酬をそれぞれ得られて得になる場合がある。もちろん相手が協調すると分かっているならば自分は裏切ったほうが得である。ゲームを繰り返すことで相手の出方を測る要素が生まれる。何回繰り返すかは固定とせず、各ターン終了後に、予め決められた確率  $p$  でゲームは終了する (確率  $1 - p$  で次のターンに続く) とする。

### 7.2.4 インターフェースの拡張と継承

ゲームを繰り返ししながら相手の戦略に対応するためには、各回で相手が選んだ選択肢を知っておく必要がある。そこでインターフェースを拡張して、各プレイヤーは `update(my_action, op_action)` というメソッドの呼び出しを通じて、結果を受け取るとする。自分の選択が `my_action` で相手の選択が `op_action` である。(自分の選択は知っているはずなので厳密には不要であるが、対称性のためこのようにしておく。なお自分の行動が確率的に決まるようなゲームでは、自分の結果を知ること必須となる。)

これに合わせて、`play_one_game` も次のように変更する。

```
Python3 1 def play_one_game(player_a, player_b):
2     act_a = player_a.play()
3     act_b = player_b.play()
4     if not valid_action(act_a): # 行動がルールに従っているかを確認
5         raise ValueError
6     if not valid_action(act_b): # 行動がルールに従っているかを確認
7         raise ValueError
8     reward_a = Reward[act_a][act_b]
9     reward_b = Reward[act_b][act_a]
10
11
12     player_a.update(act_a, act_b)
13     player_b.update(act_b, act_a) # 2行で引数の順序が異なることに注意
14
15     # print による表示は、ゲームを繰り返す場合は冗長なので除しておく
16
17     return act_a, reward_a, act_b, reward_b
```

今までのプレイヤーは相手が誰でも気にしない (行動を変えない) ので、`update(my_action, op_action)` の動作は空で良い。しかしメソッド自体は定義しないと、実行時に上記 9 行目や 10 行目でエラーとなってしまう。

3つのクラス全てにメソッドを定義して回ることもできるが、別の手段として **継承** (inheritance) を紹介する (アルゴリズム入門の教科書 [1] p. 179 のあたりを参照)。まず、プレイヤーの雛形として `Player` クラスを作る。このクラスは `name` や `play` が適切な値を返さないで、実際にゲームをプレイすることはできない。

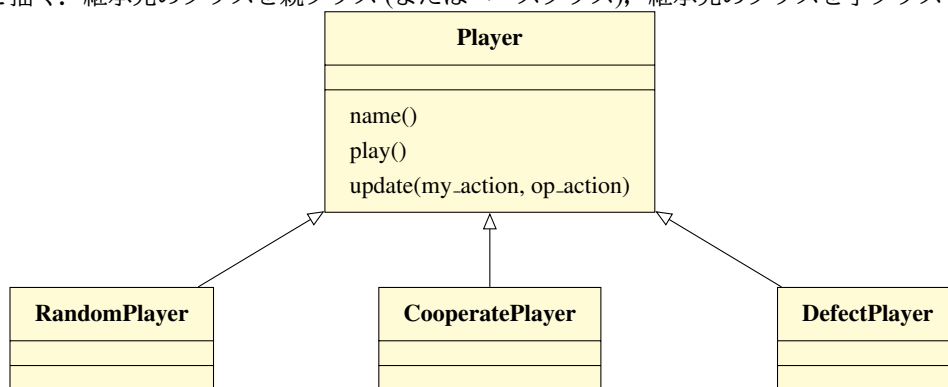
```
Python3
```

```

1 class Player:
2     def name(self):
3         pass
4
5     def play(self):
6         pass
7
8     def update(self, my_action, op_action):
9         pass

```

つづいて、既存の三つのプレイヤクラスが Player クラスを継承するようにする。これにより、Player 内で定義したメソッド update を各クラスで継承して、使えるようにする。クラス図で、継承は上下の矢印で描く。継承元のクラスを親クラス (またはベースクラス)、継承先のクラスを子クラス (サブクラス) という。



Python で親クラスを指定する方法は、クラス定義のクラス名に括弧を続けて親クラス名を指定する。赤字部分のあたりが変更点である。ここでは CooperatePlayer の定義を示すが、DefectPlayer も RandomPlayer も同様である。

**Python3**

```

1 class CooperatePlayer(Player):
2     def name(self):
3         return "CooperatePlayer"
4
5     def play(self):
6         return Cooperate

```

親クラスのメソッドは継承されるので update は各クラスで、親クラスのものが使われる。一方、name と play は子クラスで定義されているので、そちらの新しい内容が使われる。これを 上書き (overriding) という。

以上を総合して、繰り返しゲームを行う関数 play\_games は次のように作ることができる。

**Python3**

```

1 P = 1.0/8
2 def play_games(player_a, player_b):
3     sum = [0, 0] # aとbの得点
4     history = ["", ""] # aとbの選択を文字列にして保存
5     while True:
6         act_a, reward_a, act_b, reward_b = play_one_game(player_a, player_b)
7         sum[0] += reward_a

```



```

8         sum[1] += reward_b
9         history[0] += str(act_a)
10        history[1] += str(act_b)
11        if (random.random() < P): # 終了判定
12            break
13
14        print(sum, history)

```

実行例: ランダムプレイヤはたまに裏切るのでずっと協調するプレイヤよりも高得点となる。何回繰り返すかは確率的に決まるので、実行によって異なる。

Python3

```

1 >>> play_games(pc, pr)
2 [10, 31], ['000000000000', '110111000011']

```

例題

必ず前回と異なる選択をする

(金子)

必ず前回と異なる選択をするプレイヤ `AlternatePlayer` 作成せよ。

ヒント: `last_action` のようなインスタンス変数を設けて、`update` で渡される自分の指し手を記録する。それを `play` 時に参考にする。

問題

tit-for-tat (`TitForTatPlayer`)

(金子)

繰り返し囚人のジレンマゲームでは、`tit-for-tat` が有力な戦略とされている。この戦略では、基本は協調するが、相手が裏切った場合は次のターンで自分も裏切る。

この戦略を用いる `TitForTatPlayer` を作成し、今までに作ったプレイヤ 4 種類に対して、それぞれどのように振る舞うかを観察せよ。

なおこの問題の主旨は、クラスの理解も兼ねているので、資料の流れに沿って実装すること。

テスト部分は概ね次のようになるだろう。

Python3

```

1 def test_tftplayer(self):
2     player = TitForTatPlayer()
3     act = player.play()
4     self.assertEqual(Cooperate, act) # 初めは協調
5
6     player.update(Cooperate, Cooperate)
7     act = player.play()
8     self.assertEqual(Cooperate, act) # 協調同士なら協調
9
10    player.update(Cooperate, Defect) # 裏切られたら
11    act = player.play()
12    self.assertEqual(Defect, act) # 次に裏切る
13    ...

```

```
14         # 裏切り同士など、色々な組み合わせを試す
15         ...
```

---

なお、`play_games` にはインスタンスを生成した直後の `TitForTatPlayer` を与えると良い。同じインスタンスで何度も `play_games` を行う場合は、前の対戦の履歴が残っているため、最初が協調にならないことがある。

## 7.3 この章の提出課題

以下のいずれかで合格:

- 問題 (タイトル部分に ★ や optional の表記なし) を全て Python で解く
- `tit-for-tat` と問題 (タイトル部分に ★ や optional を含む) を 1 題以上解く (発展コース)

提出条件:

- 各関数に対応するテストケースが記述されていること (ユニットテストは、テンプレートの末尾の `class` にまとめて一箇所に記述する。)
- ループを含むコードは、ループ不変条件が明確にコメントで記述されていること
- 冗長なコードがないか確認し、標準的な書き方に整えられていること
- 指定した課題が、指定した関数名で作成されていること
- 提出ファイルは、1.A 節を参考にして、[jupyter から作成した](#) `weekN-(学生証番号).ipynb` とする
- 日本語 (または英語) でコメントが付記されていること: { 途中か完成品か、授業中に既に OK をもらっていればその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など、(途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む。(何も苦労がなかった場合を除き、自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

## 第 8 章

# クラスライブラリ・デザインパターン

2020-06-03 Wed

### 概要

クラスの継承関係を利用したクラスライブラリの理解方法，自分でクラス群を設計する際に指針となるデザインパターン (の一つ) を紹介する．さらに，C++ でのクラスの扱いについて触れる．今週の資料はページ数が長い，C++ の問題はオプションとしたため，必修の問題は 3 題のみである．また，今週からテストケースの作成を簡略化し，unittest 形式の代わりに後述する doctest 形式を用いる．

## 8.1 クラスとオブジェクト

Python3

```
1 class BankAccount:
2     def __init__(self, initial_balance):
3         self.__balance = initial_balance
4
5     def withdraw(self, amount):
6         if self.__balance >= amount:
7             self.__balance -= amount
8             return self.__balance
9         else:
10            return "Insufficient_funds"
11
12    def deposit(self, amount):
13        # ...
```

BankAccount
__balance
withdraw(amount) deposit(amount)

クラスとは属性と操作を同時に定義した設計図であり，それを元にオブジェクト (インスタンス) が作られる．銀行口座の例では，銀行口座のクラスを `BankAccount` と定義して，個々の口座を `BankAccount(initial_balance)` することで作成した．

■文法 Python のクラス定義で用いられる文法をいくつかまとめる．

- `class` から始まるブロックでクラスを定義する
- クラス定義内で `self.` で修飾した変数 (たとえば `self.__balance` はインスタンス変数 (C++ ではメンバ変数) である．それらは，通常の変数に似ているが，オブジェクトごとに別の変数と扱われる (口

座ごとに `self._balance` の値は異なる). またオブジェクトと同じ寿命を持つ (通常の変数は、一度関数を出ると消える).

- クラス定義内では、通常の間数定義と同様の文法で `def` からのブロックでメソッド (C++ ではメンバ関数) を定義する. メソッド内では、インスタンス変数を共通に利用可能である. メソッド呼び出しは関数呼び出しと同様だが、`a.deposit(100)` のように、(レシーバ) オブジェクトにドットを続けて実行する点が異なる. これにより、どの銀行口座を操作するかを指定する.
- `__init__` という特別なメソッドは、クラスと同名のメソッドを呼んでオブジェクトを生成した時に (たとえば `BankAccount(1000)`) 呼ばれる. 通常はここでインスタンス変数を全て初期化する. C++ でもクラスと同名のメソッドが同じ役割を果たし、`constructor`(コンストラクタ) と呼ばれる.

■性質 この演習では、(Python と C++ の共通部分をとって) 以下を前提として説明する.

- 同じクラスから作られたオブジェクトは、同じ名前のインスタンス変数 (`self._balance` など) を持つ. ただし、値は個々に異なる.
- 同じクラスから作られたオブジェクトは、同じ名前のメソッドを持ち、振る舞いも同じである.

### 8.1.1 C++ のクラス

例題

銀行口座 (C++)

(金子翻案)

銀行口座を C++ で定義せよ. ただし、`withdraw` の引数 `amount` は `m_balance` 以下であることを仮定して良い.

C++

```
1 class BankAccount {
2     int m_balance;
3 public:
4     BankAccount(int initial_balance) : m_balance(initial_balance) {
5     }
6     // BankAccount(int initial_balance) {
7     //     m_balance = initial_balance; // 書いても概ね同じ
8     // }
9     int withdraw(int amount) {
10        ...
11    }
12    int deposit(int amount) {
13        ...
14    }
15 };
```

実行確認用

C++

```
1 #include <iostream>
2 using namespace std;
```

```

3  // この辺にクラス定義を書く
4  int main() {
5      BankAccount a(10000);
6
7      cout << a.withdraw(5000) << endl; // 5000
8      cout << a.withdraw(2000) << endl; // 3000
9      cout << a.deposit(100) << endl; // 3100
10 }
```

### コンパイルと実行 (復習)

```

$ g++ -Wall bankaccount.cc 1
$ ./a.out 2
```

C++ でも Python 同様に、`class` というキーワードを用いてクラスを定義する。クラスの中には、`balance` のようなメンバ変数 (インスタンス変数) と `withdraw` のようなメンバ関数 (メソッド) を定義する。メンバ関数の中でクラスと同名のものを **コンストラクタ** と呼び、クラスの構築の際に呼ばれる (Python の `__init__` と同様の役割である)。3 行目の、コロン: から中括弧開くまでの並びは、メンバ変数をカッコ内の値で初期化する **メンバ初期化子** という構文である。メンバ変数が整数などの基本型の場合は、7 行目のように代入文で初期化しても差はない。メンバ変数がコンストラクタを持つ場合に、コンストラクタに引数を渡すにはメンバ初期化子を用いる必要がある。なおこの資料では、メンバ変数には `m_` で始まる名前を持たせた。<sup>\*1</sup> これは Python のインスタンス変数を、`_` で始まることに対応させたものであるが、Python と異なり文法上の意味はない。また Python でインスタンス変数にアクセスする場合には毎回 `self.` で修飾した。C++ で同じ修飾をするには、自信を表す予約後 `this` を用いて、`this->` と書くが、通常は省略可能である。

### 8.1.2 例外処理

十分な残高がない場合には Python の例題では文字列を返す設計としたが、C++ では **例外** を用いて実装しよう。<sup>\*2</sup>

C++ の例外処理機構では、`throw` により例外処理状態に入る。これを例外の送出と呼ぶ。また `throw` に続けて書いたオブジェクトによりどのような例外かを区別する。ここでは `<stdexcept>` で実行時例外として定義されている `runtime_error` (文字列) という例外を用いる。一旦例外が送出されると、`try-catch` ブロックに出会うまで、ブロックを抜け関数の呼び出し元に再帰的に戻る。もし対応する `try-catch` ブロックが存在しなければプログラムが終了する。以下の例で、`subtract(a,b)` は `b-a` を返す関数だが `a>b` の時は例外を送出する。例外発生後に、制御は `subtract` の呼び出し元に戻り、そこが `try` ブロック内であったので対応する `catch` 節に移動してそこから制御を再開する。

```

C++ 1 #include <iostream>
    2 #include <stdexcept>
```

<sup>\*1</sup> この命名規則は開発者の自由である。機械的に変数名 (の一部) を決めること自体にも賛否があるが、ここでは深くは立ち入らない。

<sup>\*2</sup> C++ では関数は特定の型を返す必要があるため、“整数もしくは文字列”という型を扱おうとすると煩雑になるためである。なお、先週紹介した `raise` を用いて Python で例外処理を書くことも可能である。

```

3 using namespace std;
4 int subtract(int a, int b) {
5     if (a > b)
6         throw runtime_error("a_is_greater_than_b"); // 例外を送出
7     return b - a;
8 }
9 int main() {
10    try {
11        cout << subtract(3,100) << endl; // 97
12        cout << subtract(11,10) << endl; // ここで例外が発生
13        cout << subtract(1,10000) << endl; // この行以降は実行されない
14        cout << subtract(1,10000) << endl;
15    }
16    catch (exception& e) {
17        // tryの中で何かが失敗した場合の処理
18        cout << "failed:_" << e.what() << endl;
19    }
20 }

```

例外は関数の呼び出しが1段ではなく、何段あっても try-catch ブロックまで一気に制御を移す。このため大域脱出のために用いられることもある。

## 問題

残高確認つき銀行口座の C++ での実装 (bankaccount.cc) (option) (金子翻案)

銀行口座を C++ で定義せよ。ただし、withdraw の引数 amount が m\_balance より大きい場合は `std::runtime_error("Insufficient funds")` を送出せよ。

実行確認コード:

C++

```

1 int main() {
2     BankAccount a(10000);
3
4     cout << a.withdraw(4000) << endl; // 6000
5     cout << a.withdraw(4000) << endl; // 2000
6     cout << a.withdraw(4000) << endl; // ここで例外が発生する
7     cout << a.withdraw(4000) << endl; // これらの行は実行されない
8     cout << a.withdraw(4000) << endl; //
9 }

```

```

% ./a.out
6000
2000
terminate called after throwing an instance of 'std::runtime_error'
what():  Insufficient funds

```

1  
2  
3  
4  
5

## 8.2 Python の help と docstring

Python では `type` という関数で、そのオブジェクトが何かを調べることができる。たとえば、整数 5 の型を調べると、`int` というクラスであることがわかる。さらに、`help` により説明を表示することが出来る。

```
Python3 1 >>> type(5)
2 <class 'int'>
3 >>> help(int)
```

関数 `sum` の説明も、同様に `help` で調べることも出来る。

```
Python3 1 >>> help(sum)
```

### 例題

### 説明付き長方形の面積

(金子)

演習の初めの方で作成した `rectarea` という関数に、説明を付与し、`help(rectarea)` で説明が表示されるようにせよ。

```
Python3 1 def rectarea(a,b):
2     return a*b
```

■文法: docstring Python で関数を説明するために、関数定義の `def` の行の直後に `"""` で囲んだ文字列 (複数行可) を配置する。これを docstring という。<sup>\*3</sup>

```
Python3 1 def rectarea(a,b):
2     """returns the area of ...
3     """
4     return a*b
```

`help` により確認せよ。

■文法: doctest 関数の docstring には、実行例を付与すると分かりやすい。一方で、(何らかのミスやコードの変更で) 実行例が実際の関数と異なると、読者に有害である。そこで、docstring 中の実行例を自動でテストする、doctest という機構が備わっている。それには、以下のように `>>>` という記号で実行例を表記する。なお、docstring に複数行を書く場合は、先頭を要約行にして、次を空行にすることが慣例である。

```
Python3 1 def rectarea(a,b):
2     """returns the area of a rectangle specified by width a and height b
3
4     >>> rectarea(3,5)
```

<sup>\*3</sup> <https://www.python.org/dev/peps/pep-0257/>

```

5      15
6      >>> rectarea(20,10)
7      200
8      """
9      return a*b

```

jupyter 上で doctest を実行するには、以下のように `doctest.testmod()` により行う。

```

Python3 1 import doctest
        2 doctest.testmod(verbose=True)

```

引数の `verbose=True` を消すと出力が簡潔になる。この使い方も `help(doctest.testmod)` で調べることが出来る。

問題	説明付き銀行口座	(金子)
<p>BankAccount クラスに説明を、メソッド <code>withdraw</code> に説明と <code>doctest</code> 用の実行例を付与せよ。つまり、<code>help(BankAccount)</code> や <code>help(BankAccount.withdraw)</code> で説明が表示され、実行例が <code>doctest.testmod(verbose=True)</code> で確認できるようにする。</p>		



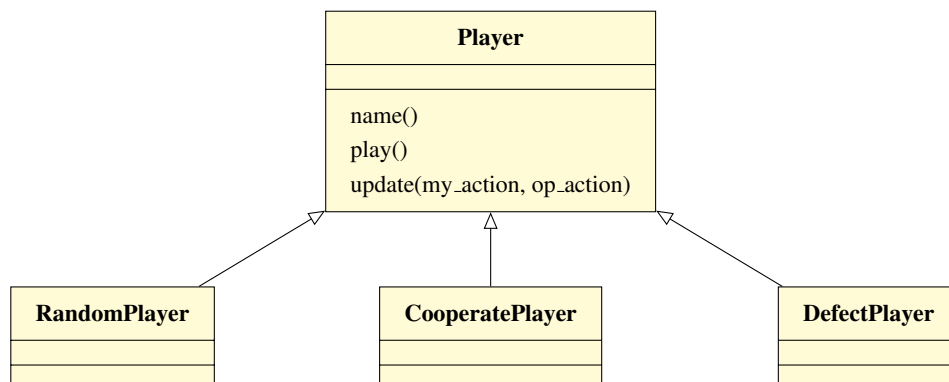
#### doctest と unittest の使い分け

doctest も unittest も、関数やメソッドの自動テストの機能を持つ。使い分けとしては、doctest には、使い方の説明として適切な (help で表示したい) 代表例を書き、unittest には網羅的なテストや複雑な状況でのテストを書くといい。

## 8.3 クラスとインターフェースの継承

クラスを定義する際に、親クラス (あるいは super class) を指定することができ、これを継承という。継承により親クラスのインスタンス変数やメソッド定義が子クラスにも受け継がれる。

ここでは、継承の機能の一部に焦点をあてて、メソッド呼び出し規約の定義の (インターフェースとしての) 継承について扱う。





まず親のクラスで、「子クラスで必ず定義してほしいメソッド」を、抽象メソッドとして定義する。抽象メソッドには、メソッド本体は書かず (docstring のみ書く)、代わりに直前に `@abc.abstractmethod` というデコレータを添える。この例では、`name` と `play` は、子クラスごとに異なる振る舞いをしてほしいので、そのように指定する。

```
Python3 1 import abc
2 class Player(abc.ABC):
3     """Player of repeated prisoners' dilemma game"""
4
5     @abc.abstractmethod
6     def name(self):
7         """Return name of the player"""
8         pass # この pass は docstring があれば不要、無い場合は必要
9
10    @abc.abstractmethod
11    def play(self):
12        """Play Cooperate or Defect"""
13        pass
14
15    def update(self, my_action, op_action):
16        pass
```

一方、`update` は、抽象メソッドではなく何もしない `pass` を本体とするメソッドとして定義した。この場合は、子クラスでそのクラス用の `update` を定義しなければ、親クラスの定義 (すなわち `pass`) が使われる。

続いて子クラスを定義する。ここでは `CooperatePlayer` の定義を示すが、`DefectPlayer` も `RandomPlayer` も同様である。Python で親クラスを指定する方法は、クラス定義のクラス名に括弧を続けて親クラス名を指定する。

```
Python3 1 class CooperatePlayer(Player):
2     def __init__(self):
3         super().__init__()
4     def name(self):
5         return "CooperatePlayer"
6     def play(self):
7         return Cooperate
```

## ■文法

- 継承元 (親): クラス名に続く括弧 ( ) 内は継承元の、親クラスを示す。
- 親クラスのコンストラクタ: `__init__` 内で `super` という特別な関数により親クラスの `__init__` を呼び出すことができる。  
通常は、子クラスの `__init__` の冒頭で `super().__init__()` を呼び出すことが適切である。
- 抽象メソッド: 子クラスは、親クラスの抽象メソッドを全て実装する。一部でも実装しない場合はインスタンス化できない。以下にエラーの例を示す。 `CooperatePlayer` で実装するべき `play` を (誤って) `pray` として実装した場合などに発生する。

```
>>> c = CooperatePlayer()
TypeError: Can't instantiate abstract class CooperatePlayer with abstract methods play
```

■継承の活用 継承を用いることで、異なるクラス (CooperatePlayer, RandomPlayer など) に共通のインターフェース (Player で定義されたメソッド名と呼び出し方) を与えることができる。

- 同じクラスから作られたオブジェクトは、異なる状態 (インスタンス変数の値) を持つことはできるが、振る舞い (メソッド定義) は共通である
- メソッドが呼ばれた時の動作 (たとえば play() が何を返すか) を変えるには、異なるクラスを作る必要がある。異なるクラス間の共通性 (たとえば play() は何らかの手を返す) を管理するために継承が有用である。

これを用いて、play\_one\_game は、引数 player\_a などが CooperatePlayer であっても RandomPlayer であっても動作するように作成することができる。

```
Python3 1 def play_one_game(player_a, player_b):
2     act_a = player_a.play()
3     act_b = player_b.play()
4     # .. 省略
```

### 8.3.1 C++ とインターフェースの継承

C++ でも同様に各種のプレイヤーのインターフェースを定めた Player クラスを定義しよう。

```
C++ 1 class Player {
2     public:
3         virtual ~Player(); // destructor
4         virtual string name()=0; // =0 は、子クラスで必ず定義されることを強制
5         virtual int play()=0; // Cooperate or Defect を選択
6         virtual void update(int, int) {
7             }
8     };
9     Player::~~Player() {
10 }
```

キーワード virtual が付与されたメソッド (メンバ関数) を 仮想関数 と呼ぶ。詳細には立ち入らないが、Python 同様に同一名称のメソッドをオブジェクト毎に振る舞いを変えたい場合には、C++ では仮想関数として定義する必要がある。仮想関数は、サブクラスで定義が上書き (override) された場合に、そちらを優先して使用するための機構である。また C++ では、継承される先祖となるクラスには仮想デストラクタを定義する必要がある<sup>\*4</sup>、3 行目と 9,10 行目でデストラクタ (~Player) の宣言と定義を行っている。

さらに、=0; で定義が終わる仮想関数を純粋仮想関数と呼ぶ。それらは、メソッドの定義がこのクラス内

<sup>\*4</sup> 厳密には必要ないケースも稀に存在するが、数々の注意点が生ずるため必要と理解しておくことを勧める。

にはなく、継承したクラスで定義されることを示す (定義なしに使用するとコンパイルエラーになる)。この Player クラスでは、name と play が純粋仮想関数である (4,5 行目)、一方、update は (純粋でない) 仮想関数である。

```
C++
1 class CooperatePlayer : public Player {
2 public:
3     string name() { return "CooperatePlayer"; }
4     int play() {
5         return Cooperate;
6     }
7 };
```

■文法 CooperatePlayer : public Player の部分は、CooperatePlayer が public に Player を継承することを示す。C++ では public でない継承もあるが、この演習では扱わない。

```
C++11
1 #include <vector>
2 // Cooperate や Reward の定義は省略する
3 vector<int> play_one_game(Player& a, Player& b) {
4     int act_a = a.play();
5     int act_b = b.play();
6     assert(valid_action(act_a));
7     assert(valid_action(act_b));
8     int reward_a = Reward[act_a][act_b];
9     int reward_b = Reward[act_b][act_a];
10    a.update(act_a, act_b);
11    b.update(act_b, act_a);
12
13    cerr << a.name() << "_v.s._" << b.name() << endl;
14    cerr << "_actions:_ " << act_a << ' ' << act_b << endl;
15    cerr << "_rewards:_ " << reward_a << ' ' << reward_b << endl;
16
17    return vector<int>({act_a, reward_a, act_b, reward_b});
18 }
```

関数 play\_one\_game の引数は、Player& a と Player クラスの参照を取るようにする。親クラスの参照 (この場合 Player クラス) は、それを継承したクラス (たとえば CooperatePlayer) のインスタンスを渡すことができる点である。

■文法 : 親クラスの参照またはポインタは、派生クラスの参照またはポインタを代入可能。

関数 play\_one\_game の戻り値は、Ruby では配列を用いたが、C++ では (配列が扱いにくい) 標準ライブラリの vector を用いた。17 行目の中括弧を用いた記法は C++11 の文法で、中括弧内の 4 つの要素を持つ vector をその場で生成する。

```
C++11
1 int main() {
2     CooperatePlayer pc;
3     DefectPlayer pd;
```

```

4     RandomPlayer pr;
5     play_one_game(pc, pd);
6     play_one_game(pc, pr);
7     play_one_game(pd, pr);
8 }

```

繰り返しゲームを行う場合の実装例を以下に示す。配列の代わりに `vector` を使っている他は、ほぼ Python のコードと対応している。

```

C++11 1 #include <random>
2 double random_value() {
3     static default_random_engine engine;
4     static uniform_real_distribution<double> distribution(0.0,1.0);
5     return distribution(engine);
6 }
7 void play_many_games(Player& a, Player& b) {
8     const double P = 1.0/20;
9     int sum[2] = {0, 0};
10    string history[2];
11    int games = 0;
12    while (true) {
13        games += 1;
14        vector<int> result = play_one_game(a, b);
15        sum[0] += result[1];
16        sum[1] += result[3];
17        history[0] += to_string(result[0]);
18        history[1] += to_string(result[2]);
19        if (random_value() < P) break;
20    }
21    cerr << a.name() << ' ' << sum[0] << ' ' << history[0] << endl;
22    cerr << b.name() << ' ' << sum[1] << ' ' << history[1] << endl;
23 }

```

1-5 行目の関数 `random_value` は C++11 の標準ライブラリで、区間  $[0.0, 1.0)$  の一様乱数を発生させる標準的な書き方。16, 17 行目の `to_string` は整数を `string` に変換する C++11 の記法。

問題

C++ 版 tit-for-tat (option) (tft.cc)

(金子)

C++ で `TitForTatPlayer` を作成し、動作を確認せよ。

## 8.4 デザイン・パターン

オブジェクト指向 (言語) でプログラムを書く利点の一つは、プログラムの再利用性を高くすることや、プログラムを組み合わせることを用意にすることである。しかし、オブジェクト指向 (言語) を学んだだけでは、そのような利点を活用する方法は自明ではなく、コツを学ぶ必要がある。[デザイン・パターン](#) (design

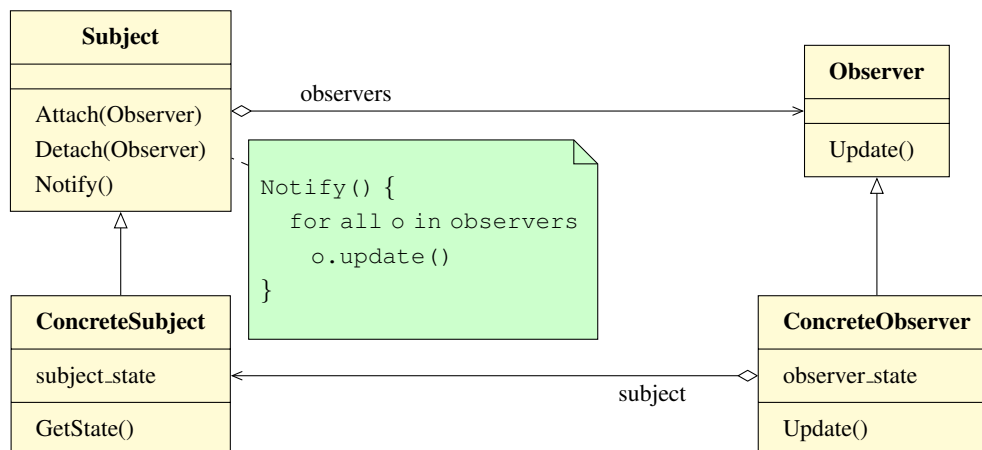
patterns)[GHJV95] は、オブジェクト指向 (言語) をもちいて見通しの良いプログラムを作るための一つの解である:

- 「このような場合には、このように設計すると良い」という経験的なものである (絶対的なルールではなく、また理論的な裏付けがあるわけではない)
- プログラム中の数個のクラスの関係についての指針である (大きなシステムの全体の設計に関する話ではないし、1 つのクラスの設計方法でもない)
- カタログである (人間が見て「ここにはこれを使おう」と思うもの)

#### 8.4.1 パターンの例: Observer パターンと時計プログラム

刻々と変化するデータを、様々な主体が観察しそれぞれ適切に行動する、というモデル化について考える。たとえば株式市場のモデル化では、株価が刻々と変化し、様々なプログラムが情報を受け取って、ユーザの画面に表示したり、自動取引を行ったりする。この時に「株価を受け取って行動するプログラム」を簡単に追加したり切り離したりできると便利である。Observer パターンは、データを保持する Subject (株式市場) と、それを監視して変化に追随する Observer (表示や取引プログラム) と役割をモデル化するものである。

株式市場は複雑なので、ここでは Subject (変化するデータ) として時計の時刻、Observer として時刻の表示などを考えてみよう。ここで、時計は全体として「時を刻む」機能と「時刻を表示する」機能を持つが、これらを切り離すことができると双方の再利用性が高まる。例えば、ユーザの好みで時刻の表示方法を取り替えたり、複数の場所に表示するような機能を、時を刻む部分に影響を与えずに実装することができる。前者を subject 側、後者を observer 側として実際に作って見よう。Subject には、以前に 7.1.2 章で作った時計クラスを流用する。



Observer パターンを説明する上記の図には 4 つのクラスが登場する。Subject と Observer は、両者の機能を抽象化したクラスである。ConcreteSubject は具体的な機能を持つ Subject で、この例では時計に相当し、時刻などのメンバー変数を保持する。ConcreteObserver は、この場合は時計の描画部分に相当し、デジタル時計やアナログ時計の機能を持つクラスが該当する。

### 8.4.2 Subject と Observer

まず、Observer パターン部分のクラス Observer と Subject を定義する:

```
Python3 1 import abc
2 class Observer(abc.ABC):
3     """abstract observer class"""
4     @abc.abstractmethod
5     def update(self):
6         """update status"""
7         pass
8
9 class Subject:
10     """abstract subject class"""
11     def __init__(self):
12         self.observers = []
13     def attach(self, an_observer):
14         """add observer to the list"""
15         self.observers.append(an_observer)
16     def detach(self, an_observer):
17         """remove observer from the list"""
18         self.observers.remove(an_observer)
19     def notify(self):
20         """call update() for all observers"""
21         for o in self.observers:
22             o.update()
```

Observer は update というメソッドを持つものである。具体的に update が何をするかは、Observer を継承したサブクラスの中で定義する。実は、Python のような言語ではこのようなクラスを実際に定義しなくても、update という名前のメソッドを持ったクラスであれば何でも Observer になり得る。

Subject の機能は、すべての observer に状態変更を通知する notify と、通知先の Observer を登録/登録解除する attach, detach である。ここでは self.observers というメンバ変数で管理することにする。

### 8.4.3 時を刻む部分の変更

先に 7.1.2 章で定義した ClockTimer は、tick() という時を進める機能と、hour などの時刻取得機能を持っていた。そのクラスを<Subject> の子クラスとなるよう、定義を少し変更する。そして、時刻が変化した場合は登録された observers の update メソッドが呼び出すために notify メソッドを呼ぶようにする。

```
Python3 1 class ClockTimer(Subject):
2     """a class manages time (self.hour, self.minute, and self.second)"""
3     def __init__(self, h, m, s):
4         super().__init__() # 追加
```

```

5         # 元の__init__の内容がここに書かれている
6     def tick(self):
7         # 元の tick の内容がここに書かれている
8         self.notify()

```

このメソッドは内部状態を変化させるので最後に `notify` を呼ぶ。また継承の際に親クラスがカバーする部分を適切に初期化するために `super().__init__()` を追加する (8.3)。

ここでは 1 秒ごとにメソッド `tick` が呼ばれるものとする。

#### 8.4.4 時刻を表示する部分

時刻を表示する部分として、ここでは単純にコンソールに文字列を書くクラス `ConsoleClock` を定義しよう。

- このクラスは、`update` が呼ばれた際に、最新の時刻を得る必要があるため、`ClockTimer` クラスのオブジェクトへの参照を持っている必要がある。そこで、インスタンス変数 `self.subject` を用意する。
- メソッド `update` は `self.subject` に入っているオブジェクトに最新の時刻を問い合わせ、表示する。表示は時分秒をコロンでつないでコンソールに書くだけとする。
- このクラスのオブジェクトが機能するためには、`ClockTimer` のオブジェクトに `attach` される必要がある。ここでは、オブジェクトが `ConsoleClock()` によって作られた際に必ず呼ばれるメソッド `__init__` 内で実現した。

Python3

```

1 class ConsoleClock(Observer):
2     """display time in console"""
3     def __init__(self, subject):
4         """create ConsoleClock and attach to subject"""
5         super().__init__()
6         self.subject = subject
7         self.subject.attach(self)
8
9     def clock_to_string(self, clock):
10        return "{:02d}:{:02d}:{:02d}".format(clock.hour, clock.minute, clock.second)
11
12    def update(self):
13        print(self.clock_to_string(self.subject))

```

実行例: まず、`ClockTimer` を作成する。ここで `timer` を `tick` すると、内部で 1 秒進むが、表示は何も起こらない。

Python3

```

1 >>> timer = ClockTimer(22, 54, 12)
2 >>> timer.tick()

```

次に、`ConsoleClock` を作成して `timer` に登録する。この状態で `timer` を `tick()` すると、連動して登録された `cc.update()` が呼ばれて、時刻が表示される。

```

Python3 1 >>> cc = ConsoleClock(timer)
        2 >>> timer.tick()
        3 22:54:14
        4 >>> timer.tick()
        5 22:54:15

```

その後、ConsoleClock を timer から切り離したとする。この状態で timer を tick() しても、表示は起こらない。

```

Python3 1 >>> timer.detach(cc)
        2 >>> timer.tick()

```

## 問題

## 複数の observer (many-observers)

(増原)

1 つの ClockTimer オブジェクトに対して複数の observer オブジェクトを作った場合でも、全ての表示が正しく行われることを確認せよ。確認後に、適切な doctest として記述すること。

## 問題

## 砂時計 (countdown)

(原案増原, 補足金子)

残り時間を測り、残り時間が無くなったらアラームを鳴らす時計 CountdownClock を作成せよ。

実装方法としては、Observer を継承して作成するとする。ConsoleClock とほとんど同じだが、Subject の tick に連動して時間を減らし、残り時間が 0 になった時にアラームを鳴らすようにすればよい。残りの時刻は、CountdownClock 側で管理し、ClockTimer は変更しないこと。適切な doctest を付与すること。

使用例:

```

Python3 1 >>> cd = CountdownClock(timer, 3)
        2 >>> timer.tick()
        3 3
        4 >>> timer.tick()
        5 2
        6 >>> timer.tick()
        7 1
        8 >>> timer.tick()
        9 !!!

```



## 8.5 C++ で class を扱う注意点

### 8.5.1 複製と参照, スマートポインタ

オブジェクトを複製した場合に何が起こるだろうか? Python の場合は, 1 つの口座に対応する銀行カードを複製したような動作になる。

```
Python3 1 >>> a = BankAccount(10000)
2 >>> a
3 # <_main_.BankAccount object at 0x102d8f5f8>
4 >>> a.withdraw(3000)
5 # 7000
6 >>> b = a # a の複製として b を作成
7 >>> b
8 # <_main_.BankAccount object at 0x102d8f5f8>
9 >>> b.withdraw(3000) # b を引き出すと
10 # 4000
11 >>> a.withdraw(3000)
12 # 1000 # a も減っている
```

C++ の場合はどうだろうか

```
C++ 1 BankAccount a(10000);
2 cout << a.withdraw(3000) << endl; // 7000
3
4 BankAccount b = a;
5 cout << b.withdraw(3000) << endl; // 4000
6 cout << a.withdraw(3000) << endl; // ???
```

C++ では, 上記の 4 行目のように書くと, 銀行のカードではなく, 同じ残高を持つ銀行口座が新しく作成される (これが望ましい動作かどうかは状況に依存する)。

```
C++ 1 BankAccount a(10000);
2 cout << a.withdraw(3000) << endl; // 7000
3
4 BankAccount &b = a; // アンパサンド記号に注意
5 cout << b.withdraw(3000) << endl; // 4000
6 cout << a.withdraw(3000) << endl; // 動作は変わるか????
```

C++ で口座に対する別名を実現するには, 参照 (上記の 4 行目) やポインタを用いる必要がある。しかし, オブジェクトの寿命に対する正確な理解が必要となり, ソースコードの管理コストが増える。本演習では, C++11 で標準ライブラリで利用可能な参照カウンタ付きスマートポインタ `shared_ptr` を勧める。

```
C++11 1 #include <memory>
2 int main() {
3     shared_ptr<BankAccount> a(new BankAccount(10000)); // 作成
```

```

4      cout << a->withdraw(3000) << endl; // 7000
5
6      shared_ptr<BankAccount> b = a; // b は a の別名
7      cout << b->withdraw(3000) << endl; // 4000
8      cout << a->withdraw(3000) << endl; // 1000
9
10     shared_ptr<BankAccount> c; // 作成時に指定しなければ、対応する口座がない
11     // c->withdraw(3000) -- segmentation fault
12     c = a; // あとから指す先を変更
13     c.reset(new BankAccount(10000)); // 新しく作った口座を指すように変更
14 }

```

3 行目の作成の読み方は, `shared_ptr<BankAccount>` が型を, `a` が変数名を, `new BankAccount(10000)` が `a` が表す実体のオブジェクトを表す. 4 行目以降の用に, メンバ関数(メソッド)を呼ぶときには, ドット(.)ではなくアロー演算子(`->`)を用いる.



ポインタとメモリ確保/解放機構を使う前に知っておくこと

文法をひと通り学べば `malloc/free` や `new/delete` を扱うので, つい(?) 使いたくなるかもしれない. しかし, 使う前には誤った使用のリスクを知っておく必要がある. 以下の規則に違反すると, プログラムの意図通りの実行は保証されない (i.e., 暴走する).

- アクセスするアドレスは正しいものであること (基本的には `new` したアドレスで, まだ `delete` されていないこと).
- 確保した方法と対応する方法で解放しなければならない (`malloc` と `free`, `new` と `delete`, `new[]` と `delete[]`).
- 確保したアドレス以外は解放してはならない (例外として `nullptr` は `delete` に渡しても無害). 二重に解放してはならない. 解放したあとはアクセスしてはならない.

なお, `free`, `delete` しないとメモリが足りなくなる場合があるが, 見積もった上で問題なければ `free`, `delete` を避けると, いくつかのリスクを回避できる.

#### 問題

#### メモリ管理 (option) (memory.txt)

(金子)

以下のコードに存在する問題点を, 具体的に指摘せよ (3 箇所上記の違反がある). 指摘を記述したテキストファイルを提出する.

C++

```

1  class Node {
2      Node *m_left, *m_right;
3  public:
4      Node() {}
5      Node(Node *l, Node *r) { m_left = l; m_right = r; }
6      ~Node() { delete m_left; delete m_right; }
7  };
8  int main () {

```

(問題 続き)

```

9   Node a;
10  Node b(new Node(nullptr, nullptr), new Node(nullptr, nullptr));
11  Node c(b);
12
13  Node x(new Node(nullptr, nullptr), new Node(nullptr, nullptr));
14  x = Node(nullptr, nullptr);
15  }

```

問題

C++ 版二分探索木 (option) (bst.cc)

(金子)

前回までに Ruby を用いて二分探索木を作成した。これを C++ でも作成せよ。

C++11

```

1  class Node {
2      int m_number;
3      shared_ptr<Node> m_left_node, m_right_node;
4  public:
5      Node(int initial_number, shared_ptr<Node> left, shared_ptr<Node> right)
6          : number(initial_number),
7            m_left_node(left),
8            m_right_node(right) {
9      }
10     int value() const { return m_number; }
11     shared_ptr<Node> left() const { return m_left_node; }
12     shared_ptr<Node> right() const { return m_right_node; }
13 };

```

実行例:

C++11

```

1  int main() {
2      int A[] = {3, 1, 4, 1, 5, 9, 2};
3      node_t t;
4      for (int a: A)
5          t = add_value(t, a);
6      list_values(t);
7      cout << endl;
8  }

```

## 問題

## C++ 版 Observer パターン (opiton) (observer.cc)

(金子)

Observer パターンを C++ で実装せよ。

C++11

```

1  int main() {
2      ClockTimer timer(22, 54, 13);
3      ConsoleClock cc(&timer);
4
5      timer.tick();
6      timer.tick();
7      timer.tick();
8  }

```

実装例は以下の通り。

C++11

```

1  class Observer {
2  public:
3      virtual ~Observer();
4      virtual void update() = 0;
5  };
6  Observer::~Observer() {
7  }
8  class Subject {
9      vector<Observer*> m_observers;
10 public:
11     virtual ~Subject();
12     void attach(Observer *observer) {
13         m_observers.push_back(observer);
14     }
15     void detach(Observer *observer) {
16         m_observers.erase(find(m_observers.begin(), m_observers.end(), observer));
17     }
18     void notify() {
19         for (Observer *o: m_observers) {
20             o->update();
21         }
22     }
23 };
24 Subject::~Subject() {
25 }

```

C++ の場合も、Python の場合と基本的に同じことを記述する。相違点として、まず `update` のようにサブクラスでは動作を変更したいメソッドには、`virtual` というキーワードを指定する。また後で継承して用いる先祖のクラスには、`virtual` 指定をしたデストラクタ (`~Observer` や `~Subject`) を定義する。Observer の一覧は、標準ライブラリである `vector` に `Observer` のポインタを入れて管理する。メソッド `detach` 内では、指定された `observer` のポインタを `m_observers` から削除する。`find` は指定範囲から

要素を探す標準関数で位置を iterator で返すものである。メソッド notify 内の for 文は、C++11 で導入された文法で、m\_observers すべてに対してブロックを実行する書式である。

C++11

```

1  #include <cstdio>
2  class ClockTimer : public Subject {
3      int m_hour, m_minute, m_second;
4  public:
5      ClockTimer(int hour, int minute, int second)
6          : m_hour(hour), m_minute(minute), m_second(second) {
7      }
8      void tick() {
9          int next_second = m_second+1;
10         int next_minute = m_minute + next_second/60;
11         int next_hour = m_hour + next_minute/60;
12
13         m_hour = next_hour % 24;
14         m_minute = next_minute % 60;
15         m_second = next_second % 60;
16
17         notify();
18     }
19     int hour() const { return m_hour; }
20     int minute() const { return m_minute; }
21     int second() const { return m_second; }
22 };
23 class ConsoleClock : public Observer {
24     ClockTimer *m_subject;
25 public:
26     ConsoleClock(ClockTimer *subject) : m_subject(subject) {
27         m_subject->attach(this);
28     }
29     ~ConsoleClock() {
30         m_subject->detach(this);
31     }
32     void update() {
33         printf("%02d:%02d:%02d\n", m_subject->hour(), m_subject->minute(),
34             m_subject->second());
35     }
36 };

```

29-31 行目の ConsoleClock のデストラクタでは、subject から detach している。このデストラクタが呼ばれる状況以降では、Subject クラスの m\_observers 内に格納されたポインタは無効となるためである（もし detach しないと次の notify で不正なメモリアクセスとなる）。なお、例にあげたコードは ConsoleClock のコピーを作ると、observer のインスタンスは必ず attach されているというこれまでの原則にあわない状態となる。実用品を作る際はコピーを禁止する（コピーコンストラクタを呼び出せないようにする）などの対処が必要である。

## 8.6 この章の提出課題

以下のいずれかで合格:

- 問題 (タイトル部分に option の表記なし) を全て Python で作成する
- 「複数の observer」と「C++ 版 Observer パターン (opiton)」を作成したうえで, (option) も含めた問題になるべく多く取り組む.

提出条件:

- 各関数に対応するテストケースが doctest で記述されていること
- 指定した課題が, 指定した関数名やクラスとメソッド名で作成されていること
- 提出ファイルは, 1.A 節を参考にして, jupyter から作成した weekN- (学生証番号) .ipynb とする
- 日本語 (または英語) でコメントが付記されていること: { 途中で完成品か, 授業中に既に OK をもらっていたらその教員名 } 学んだこと, (あれば) 他に参考にした資料や他の人の回答など, (途中で提出する場合は何がうまくゆかずに困っているなど) をコメントで盛り込む. (何も苦労がなかった場合を除き, 自作したことが分かるだけの十分なエピソードを記述すること – 授業中に教員の OK をもらった場合は簡潔で良い)

## 第9章

# 状態の保存と通信

2020-06-10 Wed

### 事務連絡

7月8日 休講 (学際科学科栃木実習)

### 概要

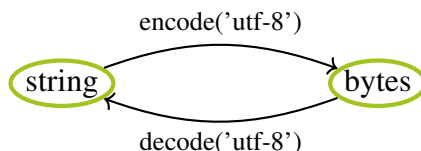
複数のプログラムが協調して動作する場合や、中断した計算を再開するためには、実行の状態を適当な表現に変換してやりとりする必要がある。テキスト形式のファイルの読み書きと、少し複雑なデータを json で表現することを紹介する。また、最終課題の準備の一部として、jupyter を離れて、コマンドラインから Python を実行する方法を紹介する。

## 9.1 事務連絡: 今後の予定

- 最終課題を来週紹介する。通常形式の課題提出は今週まで。

## 9.2 文字列とバイト列

Python や Java などでは、文字列 (string) と バイト列 bytes を区別して扱う。通常は文字列を扱えば良いが、ネットワーク通信などではバイト列が必要になる場合がある。文字列のバイト列表現を得たり、逆の変換の際には、符号化方式を指定する必要がある。



```

>>> "プログラミング".encode('utf-8')
b'\xe3\x83\x97\xe3\x83\xad\xe3\x82\xb0\xe3\x83\xa9\xe3\x83\x9f\xe3\x83\xb3\xe3\x82\xb0'
>>> "プログラミング".encode('euc-jp')
b'\xa5\xd7\xa5\xed\xa5\xb0\xa5\xe9\xa5\xdf\xa5\xf3\xa5\xb0'
  
```

```
>>> b = "プログラミング".encode('utf-8') 5
>>> b.decode('utf-8') 6
'プログラミング' 7
```

符号化方式は、日本では通常utf-8 を用いる。過去の日本では、ISO-2022-JP, EUC-JP, Shift\_JIS など使われた。同じ文字でも、符号化方式が異なると、異なった byte 列に符号化される。

byte 列は、文字で表現できるものとも限らない。任意の byte 列を ASCII で印字可能な限られた文字で表現する手法の一つがbase64 である。

## 問題

## Base64 (option)

(金子)

文字列 `s` の 'utf-8' のバイト列に対する base64 表現を得る、`myencode(s)` を実装せよ。また、同関数で変換した base64 表現から、元の文字列を得る `mydecode(s)` を実装せよ。自分の姓名(持っている人は漢字)を Base64 でエンコードして表示し、それを復号して元に戻ることを確認する doctest を付与すること。

ヒント: Python3 では base64 というモジュールが用意されている。 `import base64` して使用せよ。モジュールの使い方は `help(base64)` により調査せよ。ヒント: バイト列を base64 に変換する関数は

である。

```
>>> myencode("プログラミング") 1
'440X440t44Kw440p440f440z44Kw' 2
>>> mydecode('440X440t44Kw440p440f440z44Kw') 3
'プログラミング' 4
```


Python3 でどのようなライブラリが用意されているかは、<https://docs.python.jp/3/library/index.html> などから調査できる。

## 9.3 テキストファイルの読み書き

まず、テキストファイルを行単位で読み書きするには、`open` という関数を用いる。

```
Python3 1 with open("file-test.txt", "w") as file:
2     file.write("hello\n")
3     file.write("world\n")
```

ここで `with` で始まるブロックにおいて、`file` は、`open` したファイルを指す。ブロックを抜けると、自動で `close` される。文中の `file.write` は、指定したファイルを対象に、引数で指定された文字列の書き込みを行う。

ファイルが作成されたことを「ターミナル」を開いてコマンドで確認しよう。本日の演習では、ターミナルを多用する。ターミナルは、 というアイコンでドックに登録されている。実行例で、“\$” で始まる行は、



ユーザ (あなた) が入力するべき行の目印である。ただし, “\$” 自身はプロンプトの意図なので入力せず, その右から入力し行末でエンターキーをタイプする。使い方は, はいばーワークブックなどで確認すること。<sup>\*1</sup>

cat はファイルの中身を表示するコマンドである。プログラムの動作確認のために, 対応するコマンドを覚えておくと良い。なお, 文字 “#” とそれより右は, コメントの意図なので, 入力の必要はない。

```
$ cat file-test.txt      # テスト
hello
world
```

改行文字や, アスキーコードも含めて確認するためには, od -xc コマンドを使うことができる。

```
$ od -xc file-test.txt
00000000      6568      6c6c      0a6f      6f77      6c72      0a64
             h   e   l   l   o   \n   w   o   r   l   d   \n
00000014
```

ファイルを読み込んで表示する Python のコードは以下のようなになる。

```
Python3 1 with open("file-test.txt", "r") as file:
        2     for line in file:
        3         line = line.strip() # 改行文字を除去
        4         print(line)
```

```
hello
world
```

#### 問題

#### 二次元配列の保存 (write\_to\_file2d)

(金子)

数の配列  $x$  と  $y$  を, 1 行に 2 つの数を空白区切りで書き出す関数 `write_to_file2d(filename, x, y)` を作成せよ。先頭を 0 行目として,  $i$  行目には  $x[i]$   $y[i]$  が出力されれるとする。  $x, y$  の要素数は同じと仮定して良い。

テストは,  $y_i = \sin(x_i)$  など適当な関数を適当な刻みで設定した配列を引数に与え, 出力ファイルを gnuplot で描画する。ソースコード, 元データ (あるいはデータ生成関数) とともに, 画像をして添付せよ。

gnuplot の使い方は以下のとおりである。ファイル名が `xy.txt` であるときに描画するには, ターミナルから以下のように行う。

<sup>\*1</sup> <https://hwb.ecc.u-tokyo.ac.jp/wp/information-2/cui/terminal/>

```
$ gnuplot 1
Terminal type set to 'x11' 2
gnuplot> plot 'xy.txt' with linespoints 3
```

上記のメッセージの中で 'x11' の部分は、環境によって 'qt' などに変化する。

画面で確認後に、以下のようにタイプすると画像をファイル output.pdf に保存できる。

```
$ gnuplot 1
gnuplot> set term pdf 2
gnuplot> set out 'output.pdf' 3
gnuplot> replot 4
```

pdf 形式の保存がうまく行かない場合は、pdfcairo, png, svg などの画像形式を試すと良い。set terminal と type すると、使用可能な形式の一覧が表示される (必ずしも一般的な画像とは限らないものも含まれる)。

### 9.3.1 http での読み込み

Python では urllib.request を読み込んでおくと、open と同様にして HTTP 読み込みが可能になる。<sup>\*2</sup>

```
Python3 1 import urllib.request
2 with urllib.request.urlopen("https://www.ecc.u-tokyo.ac.jp/") as response:
3     the_page = response.read()
4     the_html = the_page.decode('utf-8')
5     print(the_html)
```

```
<!DOCTYPE html> 1
<html lang="ja"> 2
<head> 3
  <meta http-equiv="Content-Type" content="text/html;_charset=UTF-8"> 4
  <link rel="shortcut_icon" href="/favicon.ico"> 5
  <meta name="date" content="2018-04-26T12:03:09:00"> 6
  <meta name="viewport" content="width=device-width,initial-scale=1.0"> 7
  <link rel="..." 8
```

#### 例題

#### HTTP 読み込み

(金子)

指定した URL の内容を、指定したファイルに保存する関数 httpget(url, filename) を作成せよ。

<sup>\*2</sup> <https://docs.python.jp/3/howto/urllib2.html?highlight=urllib>

作成したプログラムを用いて、`http://lecture.ecc.u-tokyo.ac.jp/~ctkaneko/pl/2016/xy.txt` を保存して、`gnuplot` で描画してみよう。

なお、同様の操作は `curl` というコマンドを用いて以下のように行うことができる。また `mac` 以外では `wget` というコマンドも広く使われている。

---

```
$ curl -o filename 'http://...'
```

---

## 9.4 dict (連想配列)

Python の `dict`([マッピング](#), [連想配列](#)) は、データの組 (key, value) の組を保持し、key から value を容易に検索することができる。別の言い方をすると、配列の添字は整数だが、連想配列では整数以外にも添字のように使える。(以前二分木のところで簡単に取り扱った)。他の言語でも C++ の `unordered_map` や Ruby の [ハッシュ](#) などでも同等の機能が提供されている。

連想配列は、`{}` で構築する (配列を `[]` で構築したことに似ている)。キーに対応するデータの登録や参照は、配列と同様の文法を用いる。キーが登録済みかどうかは、キー `in` 連想配列という構文で調べることができる。

---

```
>>> dic = {} # 連想配列の構築 1
>>> dic['hello'] = 'kon-nichi-ha' # データの登録 2
>>> dic 3
{'hello': 'kon-nichi-ha'} 4
>>> 'hello' in dic # key "hello"は今登録したばかり 5
True 6
>>> 'good morning' in dic 7
False 8
>>> dic['hello'] # データの参照 9
'kon-nichi-ha' 10
>>> dic['four'] 11
Traceback (most recent call last): 12
  File "<stdin>", line 1, in <module> 13
KeyError: 'four' 14
>>> dic['four'] = 4 # 文字列以外の型も使用可能 15
>>> dic['four'] 16
4 17
```

---

### 9.4.1 配列や参照を含む dict

`dict` には、配列や別の `dict` を登録することができる。

例として、階層ディレクトリのフォルダを `dict` で表すことを考える。あるフォルダには、(フォルダでない) ファイルと、フォルダがあるので、それぞれ `"files"` と `"folders"` で表すとする

空のディレクトリを表すオブジェクトは以下のように構築できる。

```

>>> a = {}
>>> a["files"] = []
>>> a["folders"] = []
>>> a
{'files': [], 'folders': []}

```

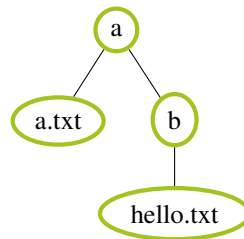
以下のように、dict の構築時に key, value を : 記号でつないで、データを指定することもできる。

```

>>> b= {"files":[], "folders":[]}
>>>
{'files': [], 'folders': []}

```

以下の図のような、階層関係を構築してみよう。



```

>>> b['files'].append('hello.txt')
>>> b
{'files': ['hello.txt'], 'folders': []}
>>> a['files'].append('a.txt')
>>> a
{'files': ['a.txt'], 'folders': []}
>>> a['folders'].append(b)
>>> a
{'files': ['a.txt'], 'folders': [{'files': ['hello.txt'], 'folders': []}]}
>>> b
{'files': ['hello.txt'], 'folders': []}

```

テスト例:

```

>>> a['files'][0]
'a.txt'
>>> a['folders'][0]
{'files': ['hello.txt'], 'folders': []}
>>> a['folders'][0]['files'][0]
'hello.txt'

```

### 9.4.2 JSON: 階層データの文字列表現

さて、階層構造を含むようなデータは、行でデータを区切るテキストファイルで表現することが難しい。そこでウェブを中心に広く使われている JSON という表現形式を紹介する。

Python ではまず `json` ライブラリを読み込む。

```
Python3 1 import json
```

これにより、Python のオブジェクトを JSON 形式の文字列に変換する `json.dumps` 関数を使えるようになる。

```
Python3 1 >>> json.dumps([3, 1, 4])
2 ' [3, 1, 4] '
```

先ほどのディレクトリ `a` のような構造を持つデータも、文字列に変換される。

```
>>> json.dumps(a) 1
'{"files": ["a.txt"], "folders": [{"files": ["hello.txt"], "folders": []}]}' 2
```

変換された文字列は、`json.loads` という手続きにより元のデータを取り出すことができる。

```
>>> json.loads(' [3, 1, 4] ') 1
[3, 1, 4] 2
```

#### 問題

#### 二分木の保存 (`write_tree`, `read_tree`)

(金子)

6.1.6 節 “内部表現の世界と使用する世界の分離” では二分木を `dict` で表現した。二分木 (たとえば Q&A で作成したもの) をファイルに保存する関数 `write_tree(filename, tree)` と読み込んで返す関数 `read_tree(filename)` を作成せよ。同じ木が復元できていることを確認せよ。

ただし、二分木の表現としては以下を使うと良い。これらは 6.1.6 章のものと同じである。

```
Python3 1 def make_node(num, left, right):
2     return {"number": num, "left": left, "right": right}
3
4 def value(tree):
5     return tree["number"]
6
7 def left(tree):
8     return tree["left"]
9
```

```

10 def right(tree):
11     return tree["right"]

```

## 9.5 numpy

Python で数値計算を行う場合は、numpy というライブラリが用意されている。<sup>\*3</sup>

以下は、numpy で 3x4 の行列を作成する例である。

```

Python3 1 import numpy as np
        2 a = np.zeros((3,4))

```

```

>>> a
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])

```

1 行目の `import numpy as np` は、`np` という別名で、`numpy` モジュールにアクセスできるようにする構文である (`numpy` を `np` という別名で扱うことが慣習のようなので、この資料でもそのようにした)。

通常 `import x` とするとモジュール `x` にアクセスできるようになる。たとえば、`import math` とすると `math.sqrt` などが使えるようになる。

■巨大なベクトル 可読性を維持するため、大きなベクトルや行列は、適宜省略して表示される。なお、下のコード例の途中の `+=1` は、全要素に対して 1 を加える演算である。

```

>>> large_array = np.zeros(10000)
>>> print(large_array)
[0. 0. 0. ... 0. 0. 0.]
>>> large_array += 1
>>> large_array
array([1., 1., 1., ..., 1., 1., 1.])

```

■数値の保存 複数の `numpy` オブジェクト (行列やベクトル) を一つのファイルに圧縮して保存することが出来る。ここでは npz という `numpy` の圧縮形式で保存する、`np.savez_compressed` を紹介する。<sup>\*4</sup> 個別にテキストファイルや `json` で保存するよりも効率が良いと期待できる。保存の際の最初の引数はファイル名で続けて、名前=変数の形式で名前を指定する。読み込みは、`np.load` で行い、帰ってきた `object` に同じ名前アクセスすると、元のデータを得られる。

```

Python3 1 np.savez_compressed("filename.npz", matrix=a, vector=large_array)

```

<sup>\*3</sup> <https://docs.scipy.org/doc/numpy/user/quickstart.html>

<sup>\*4</sup> [https://numpy.org/doc/stable/reference/generated/numpy.savez\\_compressed.html](https://numpy.org/doc/stable/reference/generated/numpy.savez_compressed.html)

```
>>> loaded_objects = np.load("filename.npz") 1
>>> loaded_objects["vector"] 2
array([1., 1., 1., ..., 1., 1., 1.]) # 保存時の large_array に相当 3
```

## 9.6 ファイル実行

これまでの演習では、標準環境として jupyter を指定し、ブラウザ上で動作するインタプリタ上で実習してきた。この方法は、比較的学习に適しているが、(実は) 標準的な実行方法ではない。

標準的には、「ターミナル」を開く、python3 コマンドにソースコードのファイル名を指定して実行する。

```
$ python3 filename.py 1
```

jupyter と比べてこの方法は、リモートの計算機を使う場合や、長時間の計算を行う場合、自動で動かす場合などに適している。

■Python3 環境への切り替え 演習端末では、default で Python2 が有効になっている (場合がある)。そこで、以下のコマンドで Python3 に変更する。

```
$ pyenv local anaconda3-4.4.0 1
```

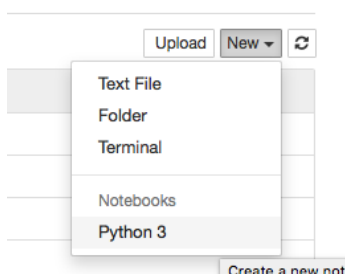
上記は一度だけ行えば、以降は (ログアウトしても) 有効である。

現在の環境を確認するには、以下のコマンドを用いる。Python 3.x と表示されればこの資料の意図通りである。

```
$ python --version 1
Python 3.6.1 :: Anaconda 4.4.0 (x86_64) 2
```

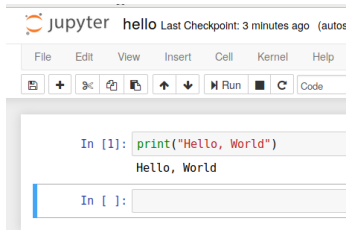
■実行例 Jupyter で開発したノートブックから Python のソースコード (.py) を出力するには以下のように行う (今後は任意のテキストエディタで代用しても良いが、テストを必ず行うこと)。

1. jupyter 上で新規ノートブックを作成する

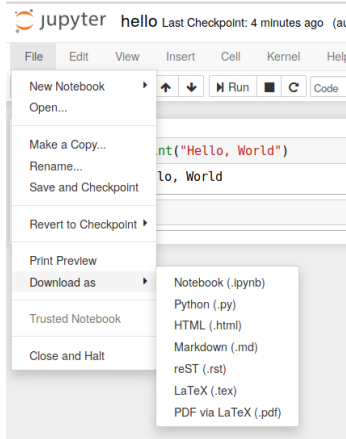


2. ノートブックを編集する

- jupyter ログの右をクリックして、ノートブックの名称を“hello”にする
- Hello, World を表示する文を書く



### 3. python ファイルに export する: File → Download as → Python(.py)



ダウンロードフォルダなど (ウェブブラウザの設定と操作による) に保存されているので, cat コマンドで確認し, つづいて, python3 コマンドで実行する.

```
$ cat hello.py 1
# coding: utf-8 2
# In[1]: 3
print("Hello, World") 4
```

```
$ python3 hello.py 1
Hello, World 2
```

#### 9.6.1 ファイルの分割

大きなプログラムの開発では, 複数のファイルに分けて開発して, 統合して使用することが多い. 一つのファイルに沢山の行数を詰め込むと見通しが悪くなるため, 機能毎に適宜分割することは良い実践である.

以下の関数が, mymessage.py というファイルに保存されていたとする.

```
Python3 1 def nice_message():
2     return "Hello, World!!!!!"
```



上記の関数を同一ディレクトリ内の `main.py` から呼ぶ方法ためには以下のように書く。

```
Python3 1 import mymessage
        2 print(mymessage.nice_message())
```

```
$ python3 main.py 1
Hello, World!!!! 2
```

この状態で、`mymessage.py` を (元気よく) 書き換えたとすると、

```
$ cat mymessage.py 1
def nice_message(): 2
    return "Hello, _Hello, _Hello, _Hello, _Hello, _World!!!!!" 3
```

`main.py` を書き換えていないにもかかわらず、実行結果が変化する。確認せよ。

```
$ python3 main.py 1
Hello, Hello, Hello, Hello, Hello, World!!!! 2
```

#### 🚫 ファイル名の注意

ファイル名は標準ライブラリと重ならないようにつけると良い。たとえば、`numpy` と関連が深い自分のプログラムがあったとして、ファイル名前、`numpy.py` ではなく `myprojectname-numpy.py` などとする。

### 9.6.2 テストの実行

ファイルに保存した Python のソースコードについて、`doctest` (8.2 章) を実行する場合は、ターミナルから次のようにタイプする。

```
$ python3 -m doctest -v filename.py 1
```

`unittest` (3.1 章) を実行する場合は、テストのクラス (`TestArea` など) とともに、テスト実行のための `unittest.main()` が以下のようにソースコードに書かれている必要がある。

```
Python3 1 if __name__ == '__main__':
        2     unittest.main()
```

この状態で `python3` を実行するとテストが行われる。

```
$ python3 filename.py 1
```

### 9.6.3 reload

複数のファイルで開発している場合、jupyter 上でもファイルの `import` が可能である。一つ注意として jupyter 上で自作ファイルを `import` 後に元ファイルを変更した場合には、そのままでは新しいファイルの内容は使われない。ファイルを読み直して新しい内容を反映させるためには、`importlib` を `import` したうえで、その `reload` という機能を用いる。

```
Python3 1 import importlib
        2 importlib.reload(modulename)
```

別の方法として、jupyter (の元の ipython) の拡張機能<sup>\*5</sup>を使い、次の行を実行しておくことでも良い。

```
Python3 1 %load_ext autoreload
        2 %autoreload 2
```

## 9.7 この章の提出課題

Python の 2 つ以上の問題について取り組み、ITC-LMS に提出する。今週はテストを書きにくいので、実行結果 (の一部) をコピーペーストで添えることで、テストの代わりとする。課題内に画像の提出などの指示がある場合は、それにも従うこと。

---

<sup>\*5</sup> <http://ipython.org/ipython-doc/stable/config/extensions/autoreload.html#module-IPython.extensions.autoreload>

## 第 10 章

# 最終課題

2020-06-17 Wed

### 概要

最終課題は、これまでの演習で練習した技術を組み合わせて多少規模の大きなプログラムをすることと、既存のプログラムとあるいは他者のプログラムと組み合わせて動かすことを主眼にする。  
なお、付録 A 章で紹介する開発ツールについて、必須とはしないが、習熟しておくことを勧める。

### 10.1 最終課題の選択肢

以下のいずれかを選んで取り組む:

1. **標準ゲームのプレイヤー**: 不完全情報ゲームの一つである潜水艦ゲームの思考部分を作る。対戦フレームワークは指定する (既存の) 実装に合わせて作ること。B 章を参照。可能な範囲で、多くの人のプレイヤーと対戦して、プレイヤーの実際の行動と自分の想定との差を把握する。
2. **標準ゲームのサーバ**: 対戦フレームワーク部分を自分で作成したコードで置き換え、他の人が作った思考部分同士を対戦させられるようにする。マップや時間制限の指定、対戦の様子のアニメーション表示や成績管理機能などがあるとよい。
3. **独自ゲーム**: 3 人程度でチームを組んで、独自のゲーム (同時着手ゲームもしくは不完全情報ゲームが望ましい) の対戦フレームワークとプレイヤーを分担して作成する。
4. **OpenAI Gym**: OpenAI Gym のエージェントを作る。C 章参照。大学院生などで、強化学習について習得済みである人向け。

### 10.2 提出期限

- 7 月 1 日の授業開始前まで:
  - スライド PDF 2,3 枚程度: プログラムの仕様, アイデアなど
  - ソースコード (1 次提出)
- 7 月 31 日:
  - ソースコード (最終提出)

- スライド (指摘があった場合)

## 第 11 章

# 最終課題作成

*2020-06-24 Wed*

## 第 12 章

# 最終課題発表

*2020-07-01 Wed*

この日までにスライド (目安:数枚) の PDF とプログラムのソースコード (できているところまで) を提出しておく。授業時間中に、構想とプログラムの説明を簡単に発表する

## 第 13 章

### (栃木実習休講)

*2020-07-08 Wed*

## 付録 A

# 開発ツール

### A.1 バージョン管理システムと最小限の git

#### 概要

継続的な開発 (あるファイル群に継続的に変更を加えるような状況) では、過去の履歴を取り出せるように、バージョン管理システムの使用を勧める (義務とはしない)。さらに、ネットワークで共有することで、自宅と大学など複数の場所での開発で連携をとったり、グループで共同開発することにも可能である。ツール類は日々新しい物が登場するので、良いツールを見極める目を養う必要がある。ここでは git を中心に紹介する。

#### 動機

- report-new2.pdf と report-0706-1.pdf はどっちが新しいんだっけ? 何を変更したんだっけ?  
➤ バージョン管理で、変更の歴史を記録する解決
- この不具合いつの間に生じたんだろう?  
➤ 古いバージョンで動かしてみて、初めて動かなくなったところが怪しい

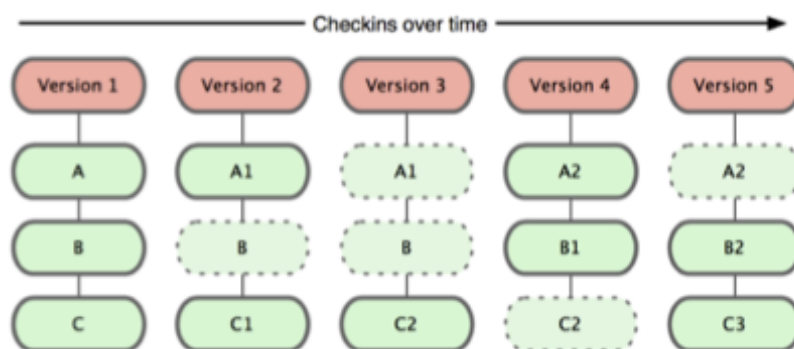


図 A.1 snapshot (ProGit 図 1.5 を転載)





## Git のインストール

この教材は ECCS の iMac 端末を前提に書かれているが、各自の PC でも利用可能と思われる。(ただし、教員の慣れている環境 ✓ 以外はサポートは難しい。)

✓ Mac OS: `brew install git` (homebrew を使っている場合)

✓ Ubuntu: `apt install git`

? Microsoft Windows: git 公式サイト <https://git-scm.com/downloads> より Windows 版をダウンロードし、同時にインストールされる gitbash というターミナル上で使う

## A.1.1 初期設定

はじまりの前に

```
% git config --global user.name "Hanako_Komaba" 1
% git config --global user.email "xxxx@g.ecc.u-tokyo.ac.jp" 2
```

はじまり: `project` というプロジェクトを作る場合、以下のような例となる。

```
% mkdir project 1
% cd project 2
% git init 3
```

既にある `programmingII` というディレクトリを登録することもできる。

```
% cd programmingII 1
% git init 2
```

## A.1.2 新規ファイルの登録

新しいファイルを作ったら/既存のファイルをバージョン管理下におくには: `git add` で登録する。add したファイルのみが図 A.1 の git の管理領域に A や B など示されるエントリを持つ。

```
% git add filename 1
```

初期バージョンの登録: (message の部分は自分のメモである。たとえば 'initial version' などとする) `commit` によりスナップショットが新規に作成され、図 A.1 左端のバージョン 1 相当の内容が記録される。

```
% git commit -m 'message' 1
```

ファイルを編集したら (変更を見る)

---

```
% git diff
```

---

1

### A.1.3 バージョン更新

ファイルを編集したら: (最新バージョンを登録) commit により, 最新版のスナップショットが作成され, 図 A.1 の時系列が右に一系列進む. (理解が進んだら: commit するファイルを選んだり, stage という概念を利用して, より細い制御をすることができる. 詳しくは Pro Gitなどを参照)

---

```
% git add files
```

---

1

```
% git commit -m 'message'
```

---

2

### A.1.4 履歴の活用

過去にどんな変更をしたか GUI で見たい

---

```
% gitk&
```

---

1

前のバージョンは何だったっけ (バージョン一覧を表示)

---

```
% git log
```

---

1

それぞれどんな変更をしたっけ

---

```
% git log -p
```

---

1

前のバージョンは何だったっけ (中身の表示)

---

```
% git show バージョン文字列: ファイル名
```

---

1

前のバージョンは何だったっけ (特定の差分の表示)

---

```
% git diff バージョン文字列 ファイル名
```

---

1

## A.2 ネットワーク経由の開発: SSH

### 目的

家から ECCS でプログラムを読み書きする。研究室のサーバとデータをやりとりする、など。

SSH を用いると、「ターミナル」で行う操作を (たとえば git コマンドも) 遠隔から行うことができる。情報系ではほぼ必須の技能である。遠隔操作を実現する場合、GUI の方が遅延が大きく手順も複雑になるため、CUI で行えるものは CUI で行う方が便利である。

ECCS 環境に ssh する方法は以下の手順を参照: <http://www.ecc.u-tokyo.ac.jp/system/outside.html#2ad4de15a21ffb51ff99a441454e9ef64>

以下、MacOS もしくは Linux が動くノート PC からアクセスする場合を例に手順を紹介する。ノート PC 上でのユーザ名が hanako、ECCS でのユーザ名が 9876543210 とする:

- (1 度のみ) ノート PC で秘密鍵と公開鍵のペアを作成する:

```
$ ssh-keygen 1
Generating public/private rsa key pair. 2
Enter file in which to save the key (/home/hanako/.ssh/id_rsa): 3
# 上記の質問は秘密鍵の保存場所 何もタイプせずにリターンするのがお勧め 4
Enter passphrase (empty for no passphrase): 5
# 上記の質問はパスフレーズ 暗記できるものを入れておくのを勧め 6
```

注意: 秘密鍵が盗まれると (ノート PC 盗難を含む)、盗んだ人はあなたの権限で ssh できる。最後の些がパスフレーズ。

- (1 度のみ) 公開鍵を eccs のホームディレクトリに登録する
  - ECCS の \$HOME/.ssh/authorized\_keys にノート PC の \$HOME/hanako/.ssh/id\_rsa.pub の行が追記されれば良い。(pub がついているファイルが公開鍵である)
  - 電子メール等で送受信してコピーペーストしても良いし、ECCS の場合は「SSH サーバ 公開鍵アップロード」 <https://secure.ecc.u-tokyo.ac.jp/eccs/keyUpload.cgi> を利用しても良い。
- ノート PC から ECCS にログイン @ の前がユーザ名、後にホスト名を書く。ECCS で利用可能なホストは 3 台ある

```
$ ssh 9876543210@ssh0-01.ecc.u-tokyo.ac.jp 1
```

パスフレーズを聞かれたらタイプする。発展: 何度もログインする際に /ssh を繰り返す際に、パスフレーズをいちいち打たないようにするには、ssh-agent や ForwardAgent などの仕組みを調べると良い。またホスト名の短縮表記やユーザ名の省略については、.ssh/config の設定方法を調べると良い。

- ファイルコピー: scp

cp コマンドと同じ文法で、リモートのファイルは `user@hostname:/path/to/file` という書式で指定する。

## A.3 リモートリポジトリの利用と共同開発

### 目的

家で作業しても大学で作業しても、同じ場所に保存したい。グループで、作業を共有したい。研究室で、卒論やソースコードを教員に見せたい。オープンソースプロジェクトを公開したい。

### 方針

共通の git リポジトリを作成し、作業場所それぞれの git の履歴を、共通リポジトリに集約する。

共有リポジトリを eccs に置く場合と、外部サービスを使う場合を紹介する。

### A.3.1 eccs に共通リポジトリを置く場合

ストーリー: 現在, [A.1.1](#) 節で紹介したように, eccs のホームの `programmingII` というディレクトリを git で管理していて, 新たに家 (もしくは手元のノート PC) でも作業をしたくなったとする。

共通リポジトリの作成:

- (一度のみ) 共通リポジトリを作業場所とは別に作成する。仮に `$HOME/git` というディレクトリに色々保管するとする。 `projectname` の部分は自分で置き換える。

```
$ mkdir $HOME/git 1
$ cd $HOME/git 2
$ git init --bare projectname.git 3
```

- (一度のみ) 既存のリポジトリに、「上流」として共有リポジトリを設定する

```
$ git remote add origin $HOME/git/projectname.git 1
$ git push -u origin master 2
```

ここで `origin` は上流の名前であり, `master` はブランチと呼ばれる開発履歴の枝分かれに対応する名前であるが, 詳しく理解するまで両者はそのままの名称を用いること。

自宅での作業場所の複製:

- (一度のみ) ssh できるようにしておく
- 複製する

```
$ git clone git+ssh://9876543210@ssh0-01.ecc.u-tokyo.ac.jp:/home/9876543210/git/project
```

- カレントディレクトリに `projectname` というディレクトリが新たにできているので、そこで作業する

通常の開発:

- 通常の開発を行い、commit する。ローカルに図 A.1 相当の履歴が記録される
- 場所を変える前に: `push` により、共有リポジトリと新しい編集履歴を共有する

```
$ git push 1
```

- 場所を変えたら、新たに編集する前に: `pull` により、共有リポジトリから新しい編集履歴をローカルに反映させる

```
$ git pull 1
```

注意: `pull` する前に手元のファイルを編集したり、手元の編集履歴を `push` する前に別の場所で `push` が行われて上流の履歴が書き換わると、編集履歴が二股に別れる。分かれた履歴の統合方法を学ぶまでは、上記の手順に厳密に従うことがお勧め。

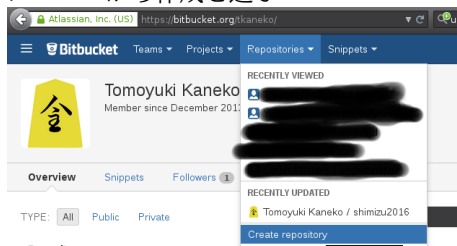
### A.3.2 外部サービスに共通リポジトリを置く場合

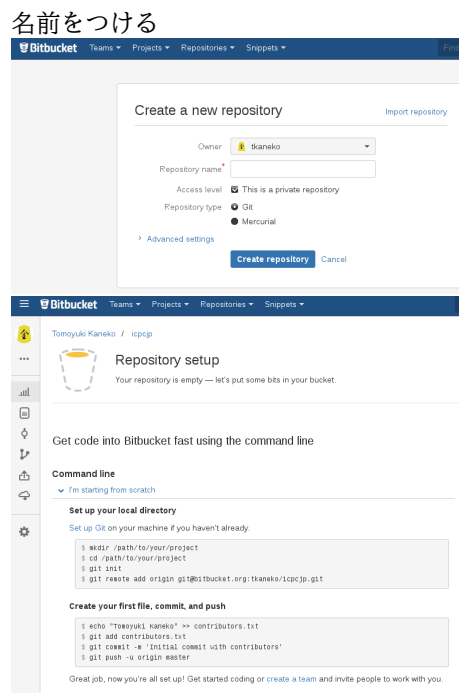
`git` リポジトリとして利用可能な外部サービスとしては `github` (<https://github.com/>) が現在最も有名である。ただし、プライベートリポジトリを作成するには手続きが必要である。

ここではプライベートリポジトリをすぐに使えることを重視して、`bitbucket` (<https://bitbucket.org/>) を例に、外部リポジトリの利用を紹介する。他の選択肢として、他に `gitlab` (<https://about.gitlab.com/>) もあるようである。教員としては、特定のサービスを勧めるわけではない。

- アカウント作成: 一般にどのサービスでもアカウントを作成する必要がある。その際には何らかの契約が行われるので、`terms and conditions` (`terms of service`, `terms of use`) などの文書を熟読すること。通常は、サービス提供側の免責が書いてあることが多いが、利用者の義務が課せられていないかどうかを注意して読むこと。一般に自分が同意したファイルは、保存しておくことを推奨。
- リポジトリの作成:

メニューから作成を選ぶ





リポジトリ作成後は、既存の作業場所から上流の登録を行う。ほぼ eccs の場合と同じだが、上流を指すアドレスが異なる。ほとんどのサービスで、https と ssh のどちらかまたは両方が使えることが多い。

```
$ git remote add origin https://user@bitbucket.org/user/projectname.git 1
$ git push -u origin master 2
```

この後の作業は、eccs の場合と同じ。

### A.3.3 複数人での開発

リポジトリは、通常は個人用に初期設定される。共同作業を行う場合は、読み書きの権限を適切な人で共有する必要がある。

共通の外部サービスを使っている場合は、ユーザごとに read, write などの権限を設定できる。eccs などファイルシステムを使う場合は、OS の「グループ」を使ったアクセス制限が可能である。ただ eccs の student グループは学生全員になってしまうため、目的に適さないことが多いと思われる。

## A.4 処理の自動化

シェルスクリプトや Makefile など (追記するかも)

## 参考書籍

Pro Git (初めの 1-2 章に目を通せば十分使える) がお勧め

- 和訳の PDF, epub など <http://progit-ja.github.io/>,
- 和訳の HTML <http://git-scm.com/book/ja>

## 付録 B

# 最終課題: 潜水艦ゲーム

この章では、潜水艦ゲームの思考部分の作成について紹介する。

### B.1 目標

- 必須: 「ランダムプレイヤー (random\_player.py) より強いプレイヤーを作る。」 「何かしら自分の考えた戦略によって行動する」
- 努力目標: なるべく強く

### B.2 環境設定と対戦

詳しくは、README.md に記載されているが、人間が AI と対戦するための方法を簡潔に記す。

1. 配布物一式をダウンロードして、適当な場所で展開する。

```
$ git clone https://github.com/tkaneko/submarine-py.git 1
$ cd submarine-py 2
```

2. ターミナルを 1 つ開いて、以下を実行する

```
$ ruby source/server.rb 2000 1
```

これによりサーバープログラムが起動し、プレイヤープログラムとの通信を待機する。

3. ターミナルをもう 1 つ (別のウィンドウまたはタブを) 開いて、以下を実行する

```
$ cd submarine-py 1
$ python3 players/random_player.py localhost 2000 2
```

これによりランダムに動く AI プログラムをサーバープログラムと接続する。

4. ターミナルをさらにもう 1 つ (別のウィンドウまたはタブを) 開いて、以下を実行する。



```
$ cd submarine-py 1
$ ruby players/manual_player.rb localhost 2000 2
```

これは、ターミナルでのキー入力により、人がゲームをプレイするプログラムである。

```
you are connected. please send me initial state. 1
please input x, y in 0 ~ 4 2
w 3
x = 0 4
y = 0 5
c 6
x = 4 7
y = 4 8
s 9
x = 2 10
y = 2 11
    0  1  2  3  4 12
----- 13
0  w3 14
----- 15
1 16
----- 17
2          s1 18
----- 19
3 20
----- 21
4          c2 22
----- 23
```

#### うまく動かない時の調査

- 3つのターミナルを注視し、エラーメッセージが出ている場合は、よく読む。一つが異常終了すると、巻き添えで全てが止まることがあるので、どれが最初かを推理する。たとえば、起動時の引数が足りない場合などはエラーメッセージから追跡できる。
- ファイアウォールが通信を止めている場合がある。(安全な環境 e.g., インターネットから切り離れた状態で) ファイアウォールを一旦停止して、動かしてみる。これで動く場合は、ファイアウォールの設定を工夫して、ファイアウォール動作状態で動くようにする。サーバ (ruby) が接続を待ち受ける許可を出したり、port 番号 (上記の例では 2000 番) での通信を許可するなどが必要と思われる。
- IPv4 v.s. IPv6: random\_player.py は IPv4 を前提に書かれているが、サーバ側は OS の設定に依存するため IPv6 で接続を待つことがある。6 月 20 日夜の commit で対策済。



#### Ruby のインストール

配布プログラムのサーバ等は Ruby を用いて書かれている。Ruby は ECCS の iMac 端末では標準で利用可能だが、各自の PC でも利用可能と思われる。(ただし、教員の慣れている環境 ✓ 以外はサポートは難しい。)

- ✓ Mac OS: brew install ruby (homebrew を使っている場合)
- ✓ Ubuntu: apt install ruby
- ? Microsoft Windows: <https://rubyinstaller.org/downloads/> からインストール

## B.3 作り方

/players/random\_player.py を参考に、戦略に基づいて行動するプレイヤーを作成する。

### B.3.1 サンプルプログラムを読む

1. README.md を読んで、全体の概要をつかむ。(ゲームのルール、ディレクトリの構造など)。ウェブブラウザで <https://github.com/tkaneko/submarine-py/blob/master/README.md> を読んでも良い。
2. doc/client\_doc.md, doc/document.md を読んで、通信の流れなどの仕様を理解する。
3. lib/player\_base.py, players/random\_player.py の RandomPlayer クラスの中身を読んで、仕様通りの動作であることを理解する。
4. 自分の AI プレイヤー players/xxxx-player.py を作る

### B.3.2 ヒント

- 人間同士でまずゲームを遊んでみる。紙で遊んでも構わないし、このプログラムを利用してもかまわない。ゲームの特徴を理解して、人間ならまずどういう作戦を立てられるかを検討する。
- プレイヤープログラムがサーバーから情報 (json) を受け取るのは、「自分の行動終了後」及び「相手の行動終了後」で、このタイミングで Player クラスの update メソッドが実行される。
- random\_player.py はゲームのルールに違反しないように実装されているが、もしゲームのルールに違反した行動をサーバーに送信するとその時点で敗北扱いになる。それを防ぐために、自分の行動をサーバーに送信する前に、ルール違反がないかどうかをチェックするようにすること。そのための補助メソッドが lib/player\_base.py に存在するので、適宜活用すると良い。

## 付録 C

# 最終課題: OpenAI Gym

この章では、最終課題のオプションの一つである、OpenAI Gym について紹介する。

強化学習は、試行錯誤を通じて学ぶ方法で、いま注目されている分野の一つである。例: <https://www.youtube.com/watch?v=TmPfTpjtdgg>

OpenAI Gym は、さまざまな問題 (環境) を統一的に扱うための、インターフェースの一つ。

### C.1 インストール

```
$ pip3 install --user gym
```

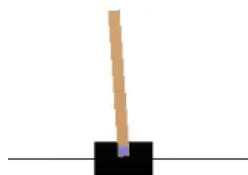
1

### C.2 動作確認

python3 を起動して `import gym` できれば成功。

```
Python3 1 import gym
        2 env = gym.make('CartPole-v0')
        3 env.reset()
        4 env.render()
```

絵が出る。



❗ import 出来ない場合

演習室の `imac` 環境では, `pip3` のインストール先 (執筆時点では `/home/.../Library/Python/3.6/lib/python/site-packages/`) を探してくれない模様. 環境変数 `PYTHONPATH` の値にそのパスを設定する<sup>a</sup>と良い. Python のソースコード中で, `os.sys.path` を設定してから, `import gym` するという方法もある.

```
Python3 1 import os
        2 os.sys.path.append("...path..to...packages...")
```

<sup>a</sup> <https://docs.python.org/ja/3/using/cmdline.html#envvar-PYTHONPATH>

## C.3 ランダムプレイ

初めに環境 `env` を作成する. `'CartPole-v0'` 以外にもさまざまな環境が用意されている (難しすぎるものもあるので注意). エージェントの行動は, `step` で伝える. 行動の選択肢は `env.action_space` で得られる. `render` で描画される. 描画は面白いが必須ではない.

```
Python3 1 env = gym.make('CartPole-v0')
        2 env.reset()
        3 for _ in range(100):
        4     env.render()
        5     env.step(env.action_space.sample())
```

## C.4 賢い行動をさせる

`env.reset()` や `env.step()` を呼び出す毎に, 状態と報酬を返り値として得られる. (上記のランダムプレイでは捨てているが), それらを適切に活用することで, 賢い行動を出来るようになる.

詳細は公式文書を参照: <https://gym.openai.com/docs/>

### C.4.1 初めの一步 (案)

`FrozenLake-v0`<sup>\*1</sup> で (よい確率で<sup>\*2</sup>) ゴールに到達するエージェントを作成せよ.

これは API に慣れる (特に `observation` を活用する) 練習なので, (学習せずに), 行動選択方法を直接プログラムに書き込んで良い.

### C.4.2 強化学習

経験から学習するには, **強化学習** を用いると良い. ニューラルネットワークを使うことが最近の流行だが, はじめは, 表 (tabular) の Q 学習または Policy Gradient を勧める. 詳しくは教科書を参照: <http://incompleteideas.net/book/the-book-2nd.html>

<sup>\*1</sup> <https://gym.openai.com/envs/FrozenLake-v0>

<sup>\*2</sup> 床は滑る/回転するので, 良い行動を選択していても運が悪いと穴に落ちる.

他の人のコードを参考にした場合は，出典と参考にした内容について，提出物内できちんと説明すること．

## 参考文献

- [1] 増原英彦 東京大学情報教育連絡会, 「情報科学入門 Ruby を使って学ぶ」, 東京大学出版会, 2010 年, <http://www.utp.or.jp/book/b306087.html> プログラミング入門」, マイナビ, 2014 年, <https://book.mynavi.jp/ec/products/detail/id=25382>
- [2] 秋葉拓哉, 岩田陽一, 北川宜稔, 「プログラミングコンテストチャレンジブック」第二版, マイナビ, 2012 年, <http://book.mycom.co.jp/book/978-4-8399-4106-2/978-4-8399-4106-2.shtml>
- [GHJV95] Erich Gamma, Richrad Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, 1995.

# 索引

`__init__`, 93

`assert`, 51  
`assertAlmostEqual`, 69  
`assertEqual`, 35

`base64`, 135  
binary search, 71

composition, 103

`deepcopy`, 62  
design patterns, 124  
`dict`, 138  
`docstring`, 118  
`doctest`, 118  
`dunder`, 99

`elif`, 23  
`else`, 23  
encapsulation, 98  
`evaluate`, 16  
expression, 16

`ffmpeg`, 61  
Fibonacci numbers, 72  
`for`, 8

Google Colaboratory, 6

`help`, 118

identity, 97  
`if`, 11, 23  
inheritance, 110  
`inorder`, 88

`jupyter`, 5

`koch`, 74

`matplotlib`, 57  
memoization, 104  
merge sort, 76

`node`, 84  
`npz`, 141  
`numpy`, 141

overriding, 111

`postorder`, 88  
`preorder`, 88  
`print`, 7  
`PYTHONPATH`, 163

recursion, 67  
`return`, 18

`slice`, 77  
statement, 7

`type`, 118

`ulimit`, 82  
`UML`, 107  
`unittest`, 34  
`utf-8`, 7, 135

`value`, 16  
variable, 17  
virtual, 121

`while`, 76

値, 16  
値渡し, 62

インスタンス, 93, 98  
インターフェース, 107  
インデント, 8

上書き, 111

オブジェクト, 97

改行, 8  
仮想関数, 121  
型, 16  
カプセル化, 98  
関数, 18

クラス, 93  
クラス図, 107

継承, 110

コッホ曲線, 74  
コンストラクタ, 116

再帰, 67  
参照, 61

式, 16  
条件分岐, 11  
真偽値, 20

スタック領域, 82

制御変数, 8  
整数, 8

節点, 84  
漸化式, 67  
  
代入, 39  
  
抽象化, 17  
抽象メソッド, 120  
  
定数, 11  
デザイン・パターン, 123  
テストケース, 35  
  
内部状態, 97  
  
二項演算子, 17  
二次元配列, 54  
二重ループ, 10  
二分木, 84  
二分探索, 71  
二分探索木条件, 89  
  
根, 84  
  
バイト列, 134  
ハッシュ, 138  
  
引数, 18  
評価, 16  
  
符号化方式, 134  
フレーム, 81  
文, 7  
  
変数, 17  
  
末尾再帰, 68  
マッピング, 138  
  
メソッド, 93, 98  
メモ化, 104  
メンバ関数, 116  
メンバ初期化子, 116  
メンバ変数, 116  
  
文字列, 22  
  
ユニットテスト, 34  
  
ライフゲーム, 57  
  
リスト内包表記, 53  
  
ループ不変条件, 41  
  
例外, 116  
連想配列, 138