
Python 演習

天野孝伸

2022 年 05 月 18 日

目次

第 1 章	はじめに	3
1.1	参考	4
第 2 章	Python の基本	5
2.1	プログラムの実行	5
2.2	ソースコードの基本構造	7
2.3	変数	8
2.4	算術演算	9
2.5	数学関数	10
2.6	キーボード入力	10
2.7	文字列フォーマット	11
2.8	第 2 章 演習課題	13
第 3 章	対話型の実行方法	15
3.1	Python の対話モード	15
3.2	IPython	17
3.3	Jupyter Qt Console	17
3.4	Jupyter Notebook	18
3.5	JupyterLab	20
3.6	様々な IDE	21
3.7	プロット	22
3.8	第 3 章 演習課題	23
第 4 章	制御構造と関数・オブジェクト	25
4.1	条件分岐 (<code>if</code>)	25
4.2	反復処理 (<code>for</code>)	27
4.3	反復処理 (<code>while</code>)	29
4.4	複雑な反復処理	31
4.5	関数	32
4.6	オブジェクト	35
4.7	第 4 章 演習課題	36
第 5 章	便利な組み込み型	41
5.1	<code>tuple</code>	41
5.2	<code>list</code>	42
5.3	<code>range</code>	44

5.4	dict	44
5.5	第 5 章 演習課題	46
第 6 章	NumPy 配列の基本	53
6.1	NumPy について	53
6.2	基本的な使い方	54
6.3	配列オブジェクト	55
6.4	生成	57
6.5	入出力	60
6.6	各種演算	64
6.7	インデックス操作	65
6.8	形状操作	67
6.9	第 6 章 演習課題	69
第 7 章	スクリプトとしての Python	79
7.1	スクリプトの基本	79
7.2	ファイルの読み書きと文字列処理	82
7.3	システムインターフェース	84
7.4	コマンドライン引数の処理	87
7.5	第 7 章 演習課題	89

注釈: この文書について

この文書は東京大学理学部地球惑星物理学科 3 年生向けの演習科目「地球惑星物理学演習」用に書かれたものです。姉妹版として [Fortran 演習](#) があります。これ以外の用途に使用することを妨げるものではありませんが、必ずしも万人向けのものではありませんのでご注意ください。Python ディストリビューションは既にインストールされているものとして扱います。また、一部の記述は Unix 系のコマンドラインが使えることを前提としています。

なお、当然のことながらこの文書の利用は自己責任でお願いします。

第 1 章

はじめに

地球惑星物理学演習ではプログラミング言語として伝統的に Fortran のみを教えてきたわけであるが、昨今の事情を考えるとこれは必ずしも時代に合っているとは言えないかもしれない。Fortran が必ずしも悪い言語というわけではないものの、科学技術計算に用いられる言語として最もポピュラーな言語の一つになった Python を学んでおくことがプラスとなることは間違いないであろう。少なくとも筆者にとっては Python は普段の研究で触れている時間が最も長い言語である。ここでは実験的な試みとして **Fortran** をある程度使えるようになったユーザー向けに Python の演習用の解説および課題を用意した^{*1}。

まずは **Python の基本** ではコマンドラインでの Python スクリプトの実行方法など Python の基礎の基礎を、**対話型の実行方法** では Python の強みの一つである様々な対話型の実行方法を簡単に紹介する。**制御構造と関数・オブジェクト** ではプログラミング言語としての Python の基礎的な文法などを解説するが、Python に関する書籍やウェブは既に多数存在するため、ここではあえて詳細には立ち入らないこととする。**便利な組み込み型** では tuple や list, dict などの便利な Python の組み込みオブジェクトについて簡単に学ぶ。これらを理解した上で、**NumPy 配列の基本** で NumPy と呼ばれるライブラリが提供する多次元配列オブジェクトの基本的な使い方を身につけよう。また、**スクリプトとしての Python** ではコマンドラインで動く Python プログラムの作り方について触れる。

この演習では **オブジェクト指向言語としての Python の使い方は扱わない** し、初学者には必ずしも必要ではないだろう。ただし、Python を使いこなすには当然どこかの段階でオブジェクト指向言語の概念を習得しておいた方がよいことは間違いないので、興味のある人は是非勉強してみたい。

なお、オフラインで勉強したい人はこのウェブページの PDF 版を以下からダウンロードすることもできる。

PDF 版: <https://amanotk.github.io/python-resume-public/python-resume.pdf>

ただし、これは機械的に変換したもので、細部の調整まではされていない。あくまでもこのウェブページが正式版である。

またサンプルコードは各章のリンクからたどることもできるが、

サンプルコード: <https://amanotk.github.io/python-resume-public/sample.tar.gz>

から一括でダウンロードしてもよい。

^{*1} 実はプログラミング言語の基礎的な概念（条件分岐や繰り返し）が理解できるようになっていれば、新しい言語を使う上で覚えなければいけないことはそれほど多くない。したがって、必ずしも Fortran でなくても C や Java などの言語の基礎的な知識があれば十分だろう。実際には Python の言語仕様としての理解よりも、強力なライブラリ群の使い方をマスターすることの方が重要である。

1.1 参考

Python に関する書籍は数多く出版されているので、紙の教科書が欲しい人は適宜好きなものを選べばよいだろう。ただし、現代ではウェブで何でも調べられるので、あまり本を購入する必要はないかもしれない。初学者向けのまとめたサイトとして以下の2つを挙げておく。

- [Python プログラミング入門 #utpython](#)
東大理学部 の講義。教材は最近公開されたようだ。結構詳しい。
- [プログラミング演習 Python](#)
京大の全学共通科目。PDF ファイルが入手できる。初学者向けでやさしい。

最近は YouTube でチュートリアル動画が多数見つかるので、(特に英語が分かる人は)YouTube で勉強するのも良いかもしれない。

第 2 章

Python の基本

まずは Python プログラムの実行方法と基本的な言語仕様を抑えよう。ただし、ここでは C または Fortran の基本的な文法は既に知っているものとして、それらと比較をしながら基本的な事項を抑えるにとどめる。ここで解説する以上の事柄については必要に応じて調べて欲しい。

参考:

- hello.py : Hello, world !
- sample1.py : ソースコードの基本構造
- sample2.py : 変数
- sample3.py : 算術演算・数学関数
- sample4.py : キーボード入力

この章の内容

- *Python* の基本
 - プログラムの実行
 - ソースコードの基本構造
 - 変数
 - 算術演算
 - 数学関数
 - キーボード入力
 - 文字列フォーマット
 - 第 2 章 演習課題

2.1 プログラムの実行

お決まりの "Hello, World !" を標準出力に表示するサンプルプログラム (hello.py) は以下の通りである。単純に print 関数に表示したい文字列を渡せばよい^{*1}。

^{*1} Python-2.x では print の形式が違っていたので古い情報を参照する場合には注意が必要。他にも細かいところに少しずつ違いがある。なお、以下のように print 関数にはカンマ区切りで複数の文字列や変数を渡すことができる。

```
print('Hi, ', 'I ', 'Love ', 'Python !')
```

リスト 1 hello.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print('Hello, World !')
```

これを実行するにはコマンドラインで

```
$ python ./hello.py
```

または

```
$ ./hello.py
```

とすればよい。2 番めの実行方法は `hello.py` が実行権限を持っていないなければならない^{*2}。実行権限を与えるには例えば

```
$ chmod 755 hello.py
```

とすればよい。

ここで C や Fortran では必須であったコンパイル作業が必要なかったことに注意しよう。Python はこれらのコンパイル型言語とは異なりインタプリタ型の言語に分類される。すなわち、コンパイルによって全てが一旦機械語に変換されてから実行されるのではなく、プログラムの実行時に一文ずつ読み込まれて、解釈され (interpret), 実行される。したがって、一般にインタプリタ型言語の実行速度はコンパイル型言語に比べて非常に遅い。Python においてもこれは正しいが、この問題は多くの場合 (少なくともある程度は) 回避することが可能である。その代わりに変数の型宣言などを事前にすることなく使うことができたりと、比較的手軽に使えるのが大きな利点である。このようにソースコードを (コンパイルすることなく) そのまま実行することができる言語をスクリプト言語と呼んだりもする。bash などのシェルも一種のスクリプト言語である。

また、Python はインタプリタとしての特徴を生かして対話的な使い方も可能である。これについては [対話型の実行方法](#) で別途紹介するが、ここではまずはコマンドラインでの実行方法に慣れておこう^{*3}。

^{*2} 実は `hello.py` の 1 行目が Python で実行せよという意味なので、実行権限さえ与えてあればこのように実行することができるのである。

^{*3} 最近では Python に関する情報の多くが Jupyter Notebook と呼ばれる環境での実行を前提としており、Python スクリプトとしての実行方法についてあまり紹介してくれていない。実際にそのような使い方を知らない人も多いようなので、ここではあえてコマンドラインでの実行方法から先に紹介する。より詳細については [スクリプトとしての Python](#) で扱う。

2.2 ソースコードの基本構造

Python のプログラムは（変数、関数、クラスなどの定義を除けば）基本的には上から順に実行されると思ってよい。ただし各行の # 以降はコメントと解釈され無視される。また、Fortran とは異なりアルファベットの大文字と小文字は区別される。

Python の大きな特徴は **構文ブロックがインデント（字下げ）で区別される** という点である。まずは以下のサンプルを見てみよう。

リスト 2 sample1.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# これはコメント行
if 1:
    # python では構文ブロックは全てインデント（字下げ）によって区切られる
    print('Hello')
    print('This will be printed out')
else:
    # ここには到達しない
    print('This will be ignored')

# ここでは sys という名前のモジュールを import する
# プログラムを実行途中で終了するには sys.exit を呼ぶ
# sys.exit に与える整数は終了コード（通常は 0 が正常終了でそれ以外は異常終了）
import sys

if 1:
    sys.exit(0)
else:
    sys.exit(-1)
```

5-11 行目が if-else による分岐である。これについては後述するが、その意味は明らかであろう。同じような分岐は C では

```
if ( 1 ) {
    printf("Hello");
    printf("This will be printed out");
else {
    printf("This will be ignored");
}
```

Fortran では

```
if ( .true. ) then
    print *, 'Hello'
    print *, 'This will be printed out'
else
    print *, 'This will be ignored'
end if
```

のように記述することになる。多くの言語では何らかのキーワードや記号 (Fortran ならば `end if` など, C ならば `{` および `}`) を用いて制御構造の区切りを表しているのに対して, Python ではインデントで区切りを付けることになっている。したがって, 例えば次のような書き方はエラーとなる。

```
if 1:
    print('Hello')
    print('error !')    # インデントがあていない
print('error !')       # 次に続く else と整合しない
else:
    print('This will be ignored')
```

すなわち, **文頭に余計なスペースなどがあるソースコードは文法的に許されない**のである。多くの言語ではあくまで人間の見やすさのためにインデントをしているのであって, 文法的には必須ではない。したがって, 書き手に依存して簡単に読みにくいコードが出来てしまう。Python ではインデントを文法として強制することで, 誰が書いても読みやすいコードになるのである^{*4}。

なお, このようなインデントを手動で行なうのはあまり賢いやり方ではないので, エディタの機能をうまく使うとよい。最近の高機能なエディタであればどれも Python を扱うための便利な機能を提供している。

また, Python には標準で様々なモジュール (ライブラリ) が用意されており, それらを組み合わせてプログラムを開発していくことになる。モジュールの使い方についての詳細な説明はここでは割愛するが, モジュールを使うにはまず `import` する必要がある。これは Fortran の `use` と同じものだと考えて良い。C では `include` に近い。

上の例では 17 行目で `sys` という名前のモジュールを `import` し, 20 行目, 22 行目で `sys.exit()` を呼んでいる。Python のプログラムは基本的には上から順に実行され最後の行に達した段階で終了するが, 何らかのエラーが発生した場合など, 途中でプログラムの実行を打ち切りたい場合には `sys.exit()` を呼べばよい。(Fortran の `stop`, C の `exit()` 関数と同じものである。)

2.3 変数

スクリプト言語である Python における変数の扱いはコンパイル型言語とは大きく異なり, 変数を使う際に型宣言をする必要がない。例えば以下のように変数を使うことができる。

リスト 3 sample2.py 抜粋

```
n = 10          # n は整数 (4byte = 32bit)
x = 3.14        # x は実数 (8byte = 64bit)

print('n = ', n)
print('x = ', x)
```

ここで変数 `n` や `x` の型を指定していないことに注意しよう。Python では明示的にデータ型を指定しなくても, `=` で数値を代入する際に (実行時に) 右辺値に応じて自動的に変数の型が選択されることになっている。したがって `n` は整数, `x` は実数となる。なお, Python3 では実数型 (浮動小数点数) はデフォルトで 8byte(C

^{*4} もちろんこれは程度の問題である。Python であっても読みにくいコードが絶対にできないわけではない。

の `double` , Fortran の `double precision` または `real(8)`) となる。一方で、整数型には決まったサイズは存在しない (値が大きくなると、必要に応じてサイズが大きくなる)。

C や Fortran ではプログラムや関数・サブルーチンの先頭で変数を宣言してから使う必要があったのに対して、いつでも好きなときに変数を定義して使うことができるので大変便利である。しかし、まだ定義していない変数を参照するとエラーが発生する。例えば先ほどの例の後に

```
print('y = ', y)
```

としても、変数 `y` が定義されていないので実行時にエラーが発生する。例えば `sample2.py` の 10 行目のコメントアウトを外して実行すると、

```
Traceback (most recent call last):
  File "sample/chap02/sample2.py", line 10, in <module>
    print('y = ', y)
NameError: name 'y' is not defined
```

のように表示されてしまう。これは意外とよくある間違いなので気をつけよう。

なお、Python ではプログラム中で書き換えのできない定数 (C の `const`, Fortran の `parameter`) は言語仕様としては提供されていない。ただし、必要な場合には実質的には定数と振る舞うような変数を用いることは可能である。

2.4 算術演算

数学的な演算は C や Fortran と共通の点が多いが、多少注意が必要である。以下に算術演算演算の使用例を示す。

リスト 4 sample3.py 抜粋

```
print(12 + 5) # 和 => 17
print(12 - 5) # 差 => 7
print(12 * 5) # 積 => 60
print(12 // 5) # 商 => 2
print(12 % 5) # 余り => 2
print(12 / 5) # 実数としての割り算 => 2.4
print(2**3) # べき乗 => 8
```

C や Fortran では整数同士で割り算 (/ 演算子) をすると返値も整数となるため、上の例で言えば $12 / 5$ は 2 となるが、Python では自動的に実数同士の割り算に変換される。代わりに `//` を整数同士の割り算をする演算子として用いる。

また `%` 演算子は C と同様に整数同士の割り算の余りを返す。Fortran では `mod` 関数に相当する。逆に `**` 演算子は Fortran と同じべき乗を返すが、C では `pow` 関数に相当する。

2.5 数学関数

数学関数を使うには `math` というモジュールを `import` すればよい。

```
import math
```

その上で、例えば $\exp(x)$ を計算するには

```
math.exp(0.5)
```

のように関数 `math.exp` を用いればよい。それ以外にも C や Fortran に組み込みで用意されているような数学関数は Python にも用意されていると考えてよいだろう。

注釈: ここでは Python 本体に組み込みのモジュールという意味で `math` を紹介した。しかし、科学技術計算で Python を使う場合にはほぼ間違いなく `numpy` というモジュールがインストールされている環境を使うことになるので、その場合には `numpy` に含まれている数学関数を用いる方が何かと便利である。とりあえずは全く同じ使い方であると考えておいて構わない。

2.6 キーボード入力

ここでコマンドラインで Python プログラムを実行する際にキーボードからの入力を受け取る方法を一つ紹介しておこう。これには組み込み関数の `input` を使う。

リスト 5 sample4.py 抜粋

```
# キーボード入力を読み込む
s = input('Input something: ')
print('Your input is : ', s)
```

input によってキーボード入力は文字列として読み込まれるので、整数や実数として用いたい場合には文字列から型を変換する関数 int もしくは float を用いる必要がある。変換された後は以下の例のように通常の整数、実数の変数として扱えば良い。

リスト 6 sample4.py 抜粋

```
# 整数に変換
a = int(input('Input integer 1: '))
b = int(input('Input integer 2: '))
print('Sum of two integers : ', a + b)

# 実数に変換
c = float(input('Input real: '))
print('Square of float : ', c*c)
```

プログラムに入力を与える方法は他にも色々と考えられるが、まずはこの一番簡単な方法を使うことにしよう。

2.7 文字列フォーマット

Python の文字列のフォーマットの仕方について簡単に触れておこう。ここではよく使われる printf 形式と format() の使い方について最低限の情報を述べるにとどめる。printf 形式は C 言語とほぼ同じで、また古い Python でも使えたことから慣れている人 (筆者のことである) が好んで使う傾向がある。format() は printf 形式の問題を解決するために導入されたという経緯があるようなので、今から覚えるのであればこちらの方がよいだろう。いずれにせよ、一気に全て覚えるようなものではなく、必要に応じて Python の公式ドキュメント等を参照して少しずつ慣れていけば十分である。

2.7.1 printf 形式

C 言語の printf で用いられるのとほぼ同じ要領で文字列のフォーマットができる。これには % を用いて、例えば以下のように用いる。

リスト 7 sample5.py 抜粋

```
s = '%s is more powerful than %s' % ('Python', 'Fortran')
print(s)
```

ここで 7 行目の文字列中の %s はフォーマットを指定子の一種で、ここでは与えられた変数を文字列としてフォーマットすることを指定している。ここでは与えられた 'Python' および 'Fortran' をそれぞれ文字列としてフォーマットし、フォーマットされた結果の文字列を変数 s に代入している。

C 言語を知っている人は C 言語の `printf` と基本的に同じフォーマット指定子が使えらと考えるとよい。例えば以下のように用いることができる。

リスト 8 sample5.py 抜粋

```
i = 2021
a = 3.1415
b = 2.7182
print("i = %-5d" % (i))
print("pi = %4.2f" % (a))
print("e = %5.3f" % (b))
```

フォーマット指定子の詳細については [公式ドキュメント](#) を参照のこと。なお、% に続く () は *tuple* オブジェクトである。ここに *dict* オブジェクトを与えることも可能であるがここでは省略する。

2.7.2 format()

Python の文字列は `format()` というメソッド^{*5} を持ち、これを用いることで文字列のフォーマットができる。以下に例を見てみよう。

リスト 9 sample5.py 抜粋

```
s = "{} is more powerful than {}".format('Python', 'Fortran')
print(s)
```

のように {} が埋め込まれた文字列の `format()` メソッドに渡した値が順に {} にフォーマットされる。デフォルトでは {} の出現順と `format()` の引数の位置 (順番) が対応しているので、この例では "Python is more powerful than Fortran" が出力される。

実際には {} には引数の位置を指定することができる。以下の例ではこれを陽に指定したもので、この結果 "Fortran is more powerful than Python" が出力されることになる。

リスト 10 sample5.py 抜粋

```
s = "{1} is more powerful than {0}".format('Python', 'Fortran')
print(s)
```

引数位置に加えて、{**引数の位置: フォーマット**} のような形で出力のフォーマットを指定することもできる。指定の仕方は `printf` 形式に似ているが、細かいところが少し違っている。以下の例は `printf` 形式の例と全く同じフォーマットで値を出力している。

^{*5} とりあえず関数と同じ意味だと思ってよい。 [ここ](#) で紹介する。

リスト 11 sample5.py 抜粋

```
print("i = {:5d}".format(i))
print("pi = {:4.2f}".format(a))
print("e = {:5.3f}".format(b))
```

なお、ここで「引数の位置」は省略されていることに注意して欲しい。

また {} に与える引数指定には、引数の順番だけでなく `format()` メソッドに与えられた引数名 (キーワード引数 参照) で指定することも可能である。詳細なフォーマット指定の方法とあわせて [公式ドキュメント](#) を参照して欲しい。

2.8 第 2 章 演習課題

2.8.1 課題 1

サンプルを実行して動作を確認しよう。

2.8.2 課題 2

標準出力に Hello, Python ! 出力するプログラムを作成しよう。

2.8.3 課題 3

標準入力から三角形の底辺 l と高さ h を読み込み、三角形の面積を表示するプログラムを作成しよう。

```
$ python kadai3.py
Input l : 5.0      # キーボード入力
Input h : 3.0      # キーボード入力
Area of triangle : 7.5
```

2.8.4 課題 4

キーボード入力で与えられた実数 x について、テイラー展開の公式

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \cdots$$

を適当な次数 (例えば 2 次とか 3 次) で打ち切りすることで $\sin x$ の近似値を求め、組み込み関数 `sin(x)` で求めた値と比較するプログラムを作成しよう。例えば $x = 0.01, 0.1, 0.2$ などについて、実行して結果を確かめること。

例えば以下のような結果が得られればよい

```
$ python kadai4.py
Input x : 0.2                                # キーボード入力
1st order = 0.2
3rd order = 0.19866666666666669
5th order = 0.19866933333333336
7th order = 0.19866933079365082
exact      = 0.19866933079506122
```

第 3 章

対話型の実行方法

これまではコマンドラインで完結した一つの Python スクリプトを実行する方法を用いてきた。ここでは Python の大きな強みの一つである対話的な実行方法について紹介しよう。ただし、対話的と一口で言っても環境は様々なので、あくまでもいくつかの代表的な実行環境を紹介するにとどめる。なお、筆者のおすすめは一昔前まで *IPython* と *Jupyter Notebook* であった。今では *JupyterLab* が両者を同時に使える大変便利な環境なので、こちらをおすすめするべきかもしれない。

参考:

[sample.ipynb](#) (ダウンロード)

この章の内容

- 対話型の実行方法
 - *Python* の対話モード
 - *IPython*
 - *Jupyter Qt Console*
 - *Jupyter Notebook*
 - *JupyterLab*
 - 様々な *IDE*
 - プロット
 - 第 3 章 演習課題

3.1 Python の対話モード

ターミナルで `python` とだけ打ってみると以下のように表示され、キーボード入力待ちの状態になるであろう。

```
$ python
Python 3.7.6 (default, Jan 8 2020, 19:59:22)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python はインタプリタ型の言語であり、一行ずつ順に実行されることは既に紹介した通りである。したがって、対話的 (interactive) に任意の Python コードを入力して実行させることができるのである。試しに `print('Hello, World !')` と入力して Enter してみると

```
>>> print('Hello, World !')
Hello, World !
```

ようになる。1 行目の Python のコードが実行され、2 行目の出力が得られたということである。

このような対話的な Python 実行環境を Python シェルとかコンソールと呼ぶことがある。Python 関係のドキュメントでは対話モードのデフォルトのプロンプト `>>>` (または `...`) に続く行が Python のコードを意味し、そうでない行は実行結果を意味することになっている。この文書でも特に断りのない限りはこの慣習に従うことにする。

もちろん 1 行では完結しない構文ブロックも入力が可能である。以下の例を見てみよう。

```
>>> for i in range(5):
...     print(i)           # 字下げ
...                        # 空行 (字下げなしで Enter)
0
1
2
3
4
```

2 行目は `for` ループの中にあるため、スペースや Tab キーで字下げをする。3 行目は字下げをせずに空白行とすることでこの構文ブロックの終端と解釈され、ループが実行されることになる。

対話モードの便利な機能として履歴の利用や、ドキュメントの参照が上げられる。対話モードでは `bash` などのシェルと同様に過去のコマンド履歴が記録されており、矢印キーや `Ctrl-p`, `Ctrl-n` で行き来することができる。ドキュメントの参照については、例えば `print` 関数のドキュメントを参照するには

```
>>> help(print)
```

と入力すると

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

(END)
```

のようにドキュメントが適当なページャ (Unix コマンドの `less` や `more` など) で表示されることになる。(大抵の場合終了するには `q` と入力すればよい。) このように、ちょっと使い方を調べたい時には対話モードのド

コメントを参照するのが便利である。

3.2 IPython

対話モードを常用するのであれば、デフォルトの対話モードをより高機能にした IPython を使うのが非常に便利である。ターミナルで `ipython` と打つと、今度は以下のように表示されるであろう。

```
$ ipython
Python 3.7.6 (default, Jan 8 2020, 19:59:22)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.12.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

今度は `In [1]:` がプロンプトとなり、入力待ちの状態になる。

基本的な使い方はデフォルトの対話モードと同じであるが、多くの便利な機能が利用できる。あまりに多いのでここでは簡単に機能を紹介するにとどめておこう。

- Unix シェルコマンドの実行 (`cd`, `ls` などがあたかもシェル上にいるかのように実行できる)
- emacs モード・vi モード (他にも細かなキーバインド設定ができる)
- シンタックスハイライト (コードの色付けをしてくれる)
- 自動インデント (構文ブロックに入ると自動的にインデントされる)
- 構文ブロックの複数行編集 (構文ブロックの中では前の行に戻って編集ができる)
- `Out [N]` の形での出力結果の参照 (出力されたオブジェクトを後から参照できる)
- タブキーでの補完 (適宜補完および補完候補を表示してくれる)
- `?` でドキュメント参照 (`help(print)` の代わりに `print?` としてもよい)
- 様々なマジックコマンド

便利なマジックコマンドが多数存在するが、これについて紹介し始めるときりがないので各自で調べてほしい。特に `%edit` や `%hist` などが便利である。

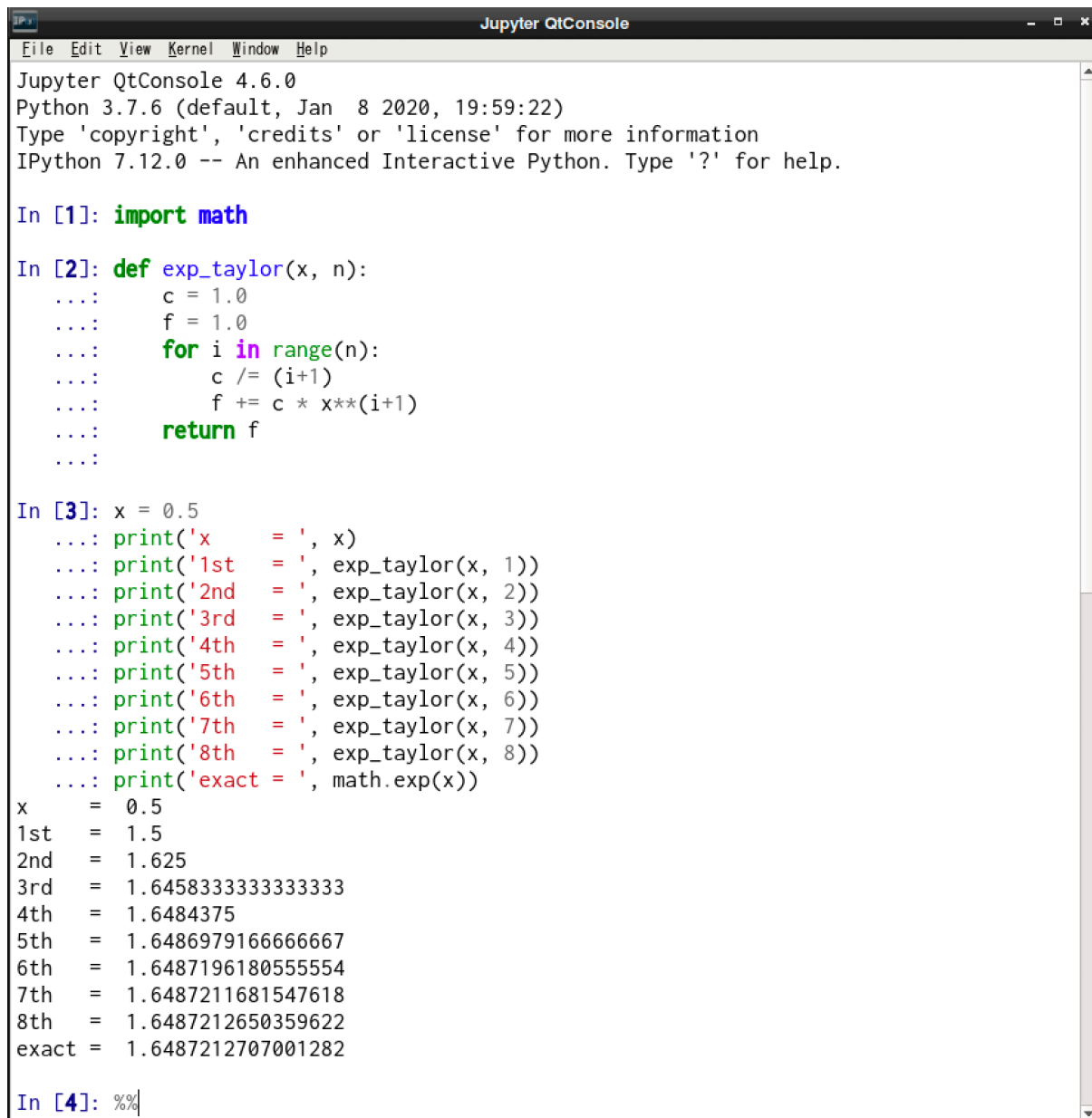
3.3 Jupyter Qt Console

IPython のコンソールと機能的にはほぼ同じであるが、専用の GUI を持つ Jupyter Qt Console というアプリケーションも存在する。ターミナルで

```
$ jupyter-qtconsole
```

のようにコマンドを実行するとこのアプリケーションが起動する。

筆者は個人的には IPython のコンソールの方が使いやすいと感じるが、こちらの方が好みの人もあるかもしれない。図をプロットする場合に別ウィンドウでなく、このウィンドウ内に埋め込むことができるという IPython にはない利点がある。(しかし、そのような使い方であれば次に紹介する *Jupyter Notebook* の方が使いやすい。)



```

Jupyter QtConsole 4.6.0
Python 3.7.6 (default, Jan  8 2020, 19:59:22)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.12.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import math

In [2]: def exp_taylor(x, n):
...:     c = 1.0
...:     f = 1.0
...:     for i in range(n):
...:         c /= (i+1)
...:         f += c * x**(i+1)
...:     return f
...:

In [3]: x = 0.5
...: print('x      = ', x)
...: print('1st   = ', exp_taylor(x, 1))
...: print('2nd   = ', exp_taylor(x, 2))
...: print('3rd   = ', exp_taylor(x, 3))
...: print('4th   = ', exp_taylor(x, 4))
...: print('5th   = ', exp_taylor(x, 5))
...: print('6th   = ', exp_taylor(x, 6))
...: print('7th   = ', exp_taylor(x, 7))
...: print('8th   = ', exp_taylor(x, 8))
...: print('exact = ', math.exp(x))
x      = 0.5
1st    = 1.5
2nd    = 1.625
3rd    = 1.6458333333333333
4th    = 1.6484375
5th    = 1.6486979166666667
6th    = 1.6487196180555554
7th    = 1.6487211681547618
8th    = 1.6487212650359622
exact  = 1.6487212707001282

In [4]: %%

```

図1 Jupyter Qt Console

3.4 Jupyter Notebook

ターミナルで

```
$ jupyter-notebook
```

を実行してみよう。デフォルトのブラウザが新しく立ち上がり、下のようなページが表示されるだろう。ここで右側の「New」から「Python 3」を選択してみよう。

この状態で表示されるのが下の図のような notebook と呼ばれるページで、各セルに Python のコードを入力して実行することができる。セルにフォーカスが当たった状態で **Shift+Enter** を入力すると、そのセルのコードを実行し、出力があればそれがセルの下に表示される。なお、セルには Markdown と呼ばれる形式で任意のテキストや数式などを入力し表示することができる。

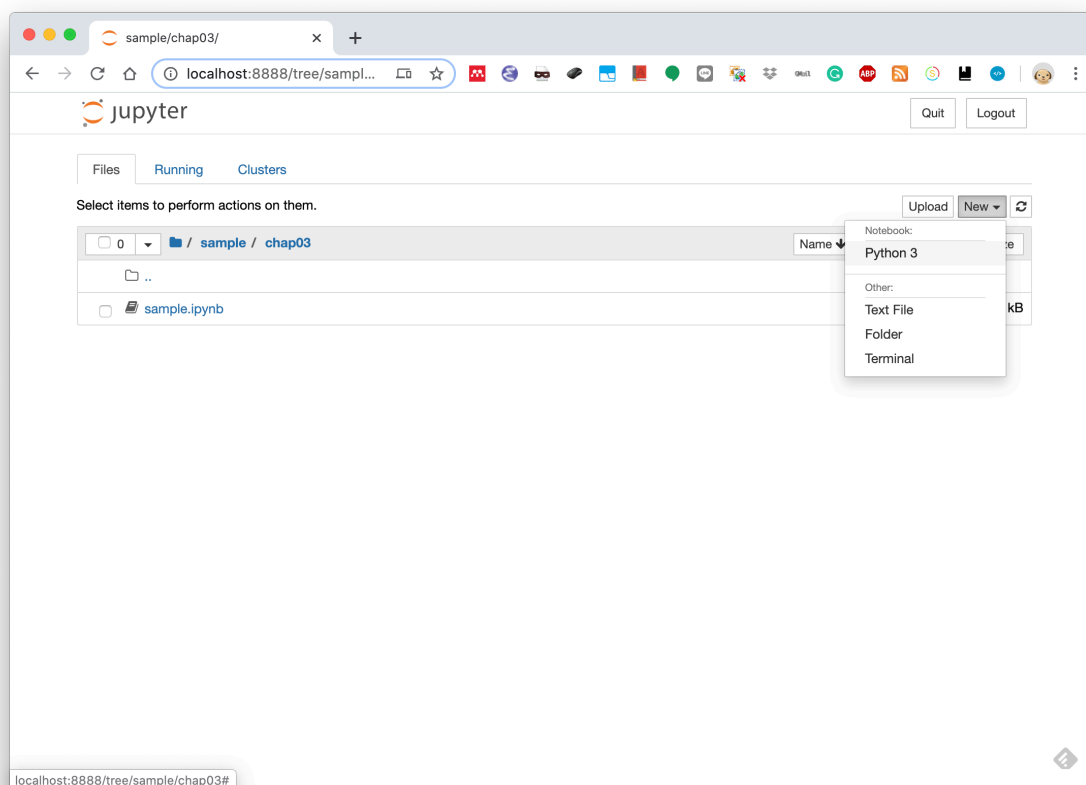


図2 Jupyter Notebook の起動画面

注釈: Jupyter Notebook は手軽に Python コードを実行できる非常に便利な環境である。近年になってこの環境があまりにも普及したので、「Python の実行環境 = Jupyter Notebook」と考えている人もいるかもしれない。しかし、Notebook 上で試行錯誤しながらデータの解析作業などをしていると、セルの実行順序や変数や関数の定義などの関係が分かりづらくなってしまふ（というかほとんど分からなくなる）。「出来た」と思っても次に開いた時には（変数が宣言されていなかったり、セルの順序がバラバラだったり）同じ結果が再現できないということが多々発生する。こうならないためには、メニューの「Kernel」→「Restart & Run All」で結果が変わらないことを確認するのが大変重要である。また、ある程度のまとまった処理は Python のソースコード（.py ファイル）として分離して、Notebook からは import して使うようにする方が読みやすく、スッキリとするであろう^{*1}。

なお、新たにブラウザを立ち上げたくない場合^{*2}には

```
$ jupyter-notebook --no-browser
```

とオプションを付けて起動をすればよい。デフォルトではブラウザで <http://localhost:8888/> にアクセスすれば Jupyter のサーバーに接続することができる。8888 はデフォルトのポート番号であるが `--port` オプションで変更することもできる。

^{*1} これはモジュールを自作することを意味しており、少し高度なトピックである。

^{*2} 例えばリモートサーバーに ssh 接続している場合など。

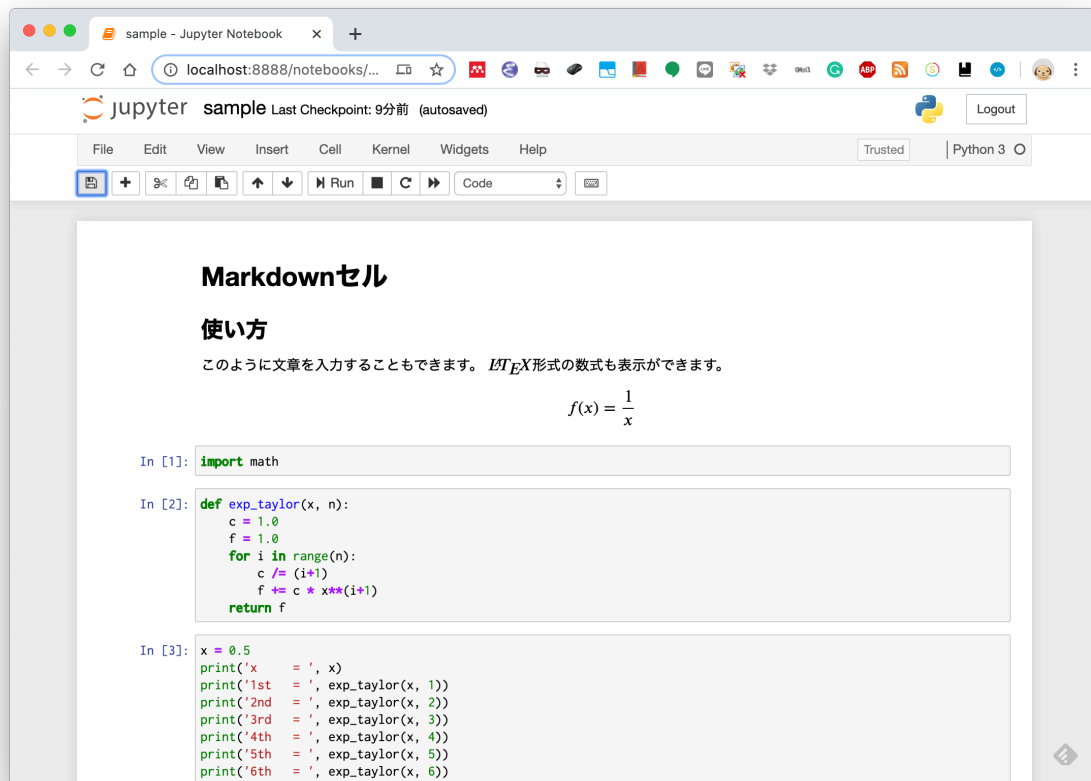


図3 Jupyter Notebook の使用例

3.5 JupyterLab

JupyterLab は Jupyter Notebook の次世代版という位置付けのウェブベースの統合開発・解析環境と呼べるものである。起動するには Jupyter Notebook と同じように

```
$ jupyter-lab
```

とすればよい。

デフォルトで Jupyter Notebook はもちろん、シェルのコンソール、IPython コンソール、テキストエディタ、ファイルブラウザなどが使えるようである。タブを分割することもできるなど非常に高機能な環境である。今から使い始める人は最初からこれを使ってもよいかもしれない。

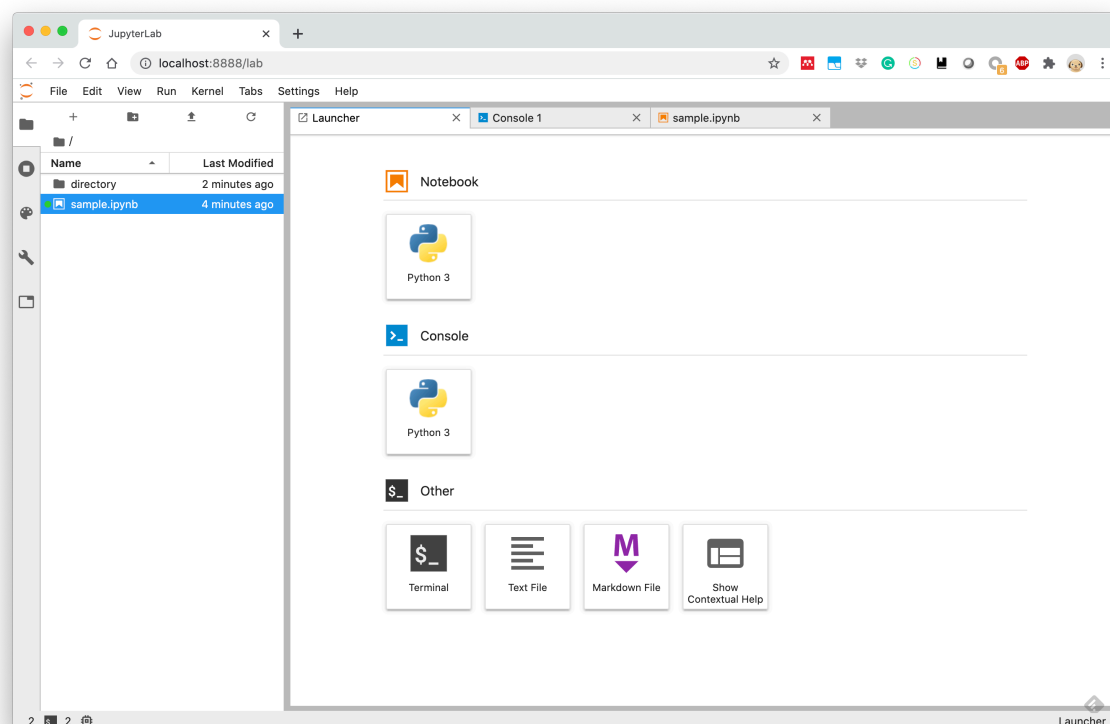


図 4 JupyterLab の画面

3.6 様々な IDE

多くの Python の統合開発環境 (IDE) でも対話的な実行方法を備えているようだ。内部的に IPython のシェルを使っていたり、Jupyter Notebook のようにセル単位で実行できたりする。変数ブラウザや補完、デバッガ機能などもあり、使いこなすことができればかなり有用そうなので、興味のある人はぜひトライしてみて欲しい。以下にいくつか例を挙げておこう。

- Spyder

Anaconda に付属する IDE。使いこなせば便利そう。

- PyCharm

高機能な IDE らしい。有償版と無償版が存在する。

- Visual Studio Code

エディタでもあり、開発環境でもある。Python extension に Jupyter Notebook を使える機能が追加されている。

3.7 プロット

IPython や Jupyter Notebook を使うと対話的にデータのプロットができる。Python では matplotlib というデファクトスタンダードに近いライブラリを使ってプロットをすることが多い。例えば以下のようにすればよい^{*3}。

```
# numpy
import numpy as np

# matplotlib のおまじない
from matplotlib import pyplot as plt

x = np.linspace(0.0, np.pi, 100)
y = np.cos(x)**2
plt.plot(x, y)
```

これを Jupyter Notebook で実行すると以下のように図が Notebook に埋め込まれた形で表示される。なお、ここで用いている numpy モジュールについては [NumPy 配列の基本](#) で紹介する。

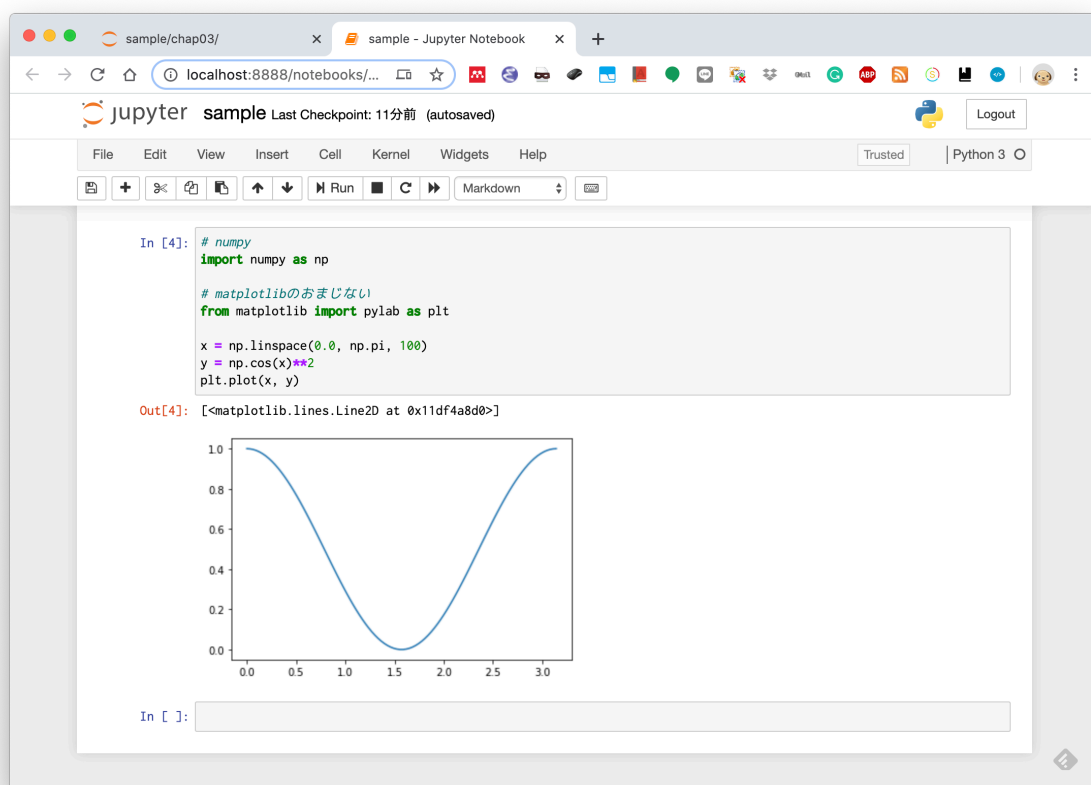


図 5 Jupyter Notebook でのプロット

^{*3} matplotlib の使い方を説明し始めると長くなるのでここでは割愛する。なお、Python から使えるプロットライブラリはこれ以外にも多数存在し群雄割拠状態である。matplotlib が最も広く使われていることは間違いないので、まずはこれを使ってみることをオススメするが、興味のある人は plotly, bokeh, holoviews などについても調べてみると面白い。

3.8 第3章 演習課題

3.8.1 課題 1

ターミナルから Python の対話モードを起動し、変数の定義や四則演算、`print()` 関数を使った出力、ヒストリ機能などを実際に試してみよう。

3.8.2 課題 2

ターミナルから IPython を起動し、補完機能や複数行の編集 (ループ処理など) などを実際に試してみよう。Python のデフォルトの対話モードと比べて便利なのが実感できるであろう。また Jupyter Qt Console との使用感の違いや、IPython の便利な機能を各自で調べてみよう。

3.8.3 課題 3

JupyterLab または Jupyter Notebook でサンプルを開き、実際に動作を確認しよう。デフォルトの対話モードや IPython と比べて使用感の違いを実感しよう。また、JupyterLab の Console でも IPython を使ってみよう。

3.8.4 課題 4 [†]

各種 IDE について調べ、興味のあるものを実際にインストールし使ってみよう。なお、Spyder は anaconda によって Python をインストールした場合にはデフォルトでインストールされている。

第 4 章

制御構造と関数・オブジェクト

ここでは `if` による条件分岐, `for` や `while` による反復処理, さらには関数の基本的な使い方や, オブジェクトの基本についても学ぼう.

参考:

[sample.ipynb](#) (ダウンロード)

この章の内容

- 制御構造と関数・オブジェクト
 - 条件分岐 (`if`)
 - 反復処理 (`for`)
 - 反復処理 (`while`)
 - 複雑な反復処理
 - 関数
 - オブジェクト
 - 第 4 章 演習課題

4.1 条件分岐 (`if`)

Python でも基本的な考え方は他の多くのプログラミング言語と同じで, なんらかの値の真偽に応じて処理の流れを分岐する. 例えば `status` という変数の真偽に応じて処理を分岐するには

```
>>> status = True
>>>
>>> # if による分岐
>>> if status:
...     print('status is True') # インデントが1段下がる
... else:
...     print('status is False') # ここの1段下がる
status is True
```

のように記述すればよい. ここでインデントが構文ブロックの区切りであったことを思い出して欲しい. `if`

に限らず至るところでインデントが必要になるので注意しよう。

なお `else` 節は必要なければ省略することが可能である。C や Fortran と異なり、条件式の部分にカッコが必要ないことに注意しよう。（ただしカッコがあっても文法的には誤りではない。）さらに、`if` を入れ子にするなどのより細かい制御の例として以下を見てみよう。

```
>>> # 入れ子にするなどより細かい制御
>>> a = 1
>>> b = 2
>>> c = 3
>>>
>>> if a:
...     print('a = ', a)
...     # if を入れ子にする
...     if a + b > c:
...         print('a + b > c')
...     elif a + b < c:
...         print('a + b < c')
...     elif a + b == c:
...         # さらに入れ子にする
...         if c >= 0:
...             print('a + b = c and c >= 0')
...         else:
...             print('a + b = c and c < 0')
...     else:
...         # pass を使って「何もしない」ことを明示することもできる
...         pass
>>> # トップレベルの if に対応する else は省略
a = 1
a + b = c and c >= 0
```

なお 21 行目の `pass` は「何もしない」という意味の文である。 `else` 節は省略できるので、わざわざここに `pass` は使う必要はないのだが、場合によっては「何もしない」ことを明示しておくことでプログラムが理解しやすくなることもある。

ここで分岐に使う条件式についてもコメントしておこう。Fortran では条件式は真偽値 (logical 型の変数、または `.true.`, `.false.` のいずれか) に限られていた。それに対して、Python の場合は C に近づいている。すなわち、Python 組み込みの `bool` 型である `True` や `False`（これらは Python の予約済みキーワードで大文字小文字は区別される）だけでなく、任意の型の変数を使うことが出来る。詳細は省くが、とりあえずは C と同様に整数型では 0 が偽とみなされ、それ以外は真とみなされるとおぼえておけばよいだろう。また複雑な条件判定には論理演算子である `and`, `or`, `not` を組み合わせることになる。以下のテーブルには条件判定によく使われる演算子およびその組み合わせの代表的な例を示す。なお、Python は C や Fortran よりも演算子が多い (例えば `is` 演算子) のだが、ここではそれには触れないことにする。

表 1 論理演算子

Python	C	Fortran
<code>A > B</code>	<code>A > B</code>	<code>A > B</code>
<code>A >= B</code>	<code>A >= B</code>	<code>A >= B</code>
<code>A < B</code>	<code>A < B</code>	<code>A < B</code>
<code>A <= B</code>	<code>A <= B</code>	<code>A <= B</code>
<code>A == B</code>	<code>A == B</code>	<code>A == B</code>
<code>A != B</code>	<code>A != B</code>	<code>A /= B</code>
<code>A > B and A > C</code>	<code>A > B && A > C</code>	<code>A > B .and. A > C</code>
<code>A > B or A > C</code>	<code>A > B A > C</code>	<code>A > B .or. A > C</code>
<code>not A > B</code>	<code>! (A > B)</code>	<code>.not. A > B</code>

以下は論理演算子を使って様々な条件を組み合わせた例である。演算子の組み合わせにはあらかじめ決められた優先順位に応じて順次評価されるが、カッコで明示的に示したほうが分かりやすい。

```
>>> if a < b and a < c:
...     print('a is smaller than both b and c')
a is smaller than both b and c

>>> if a < b or a < c:
...     print('a is smaller than either b or c')
a is smaller than either b or c

>>> if (a == 1 and a + b == c) or not (a == 1 and b == 2 and c == 3):
...     print('example of complex conditional')
example of complex conditional
```

ちなみに Python には C の `switch`、Fortran の `select` に対応する構文は存在しないので、`if` と `elif` で地道に条件判定をつないでいくことになる。

4.2 反復処理 (`for`)

Python における反復処理（ループ）の最も基本的な形は以下のようなものである。

これを実行すると 0, 1, 2, 3, 4 が順に出力される。もう少し細かい制御をするのであれば

```
>>> for i in range(5):
...     print(i)
0
1
2
3
4
```

```
for i in range(start, stop, increment):
    print(i)
```

ようになる。これによって、ループ変数 i が $i = \text{start}$ から increment ずつ変化しながら $i < \text{stop}$ を満たす限りループの中身が実行される。ここで start , stop , increment は全て整数（または整数型の変数）である必要がある。

これは C なら

```
for(int i=start; i < stop ;increment++) {
    printf(i);
}
```

Fortran なら

```
do i = start, stop-1, increment
    print *, i
end do
```

に対応する。なお `range(5)` のように 1 つしか数値を与えなかった場合は、与えられた数値は上記の `stop` と解釈され、`start = 0, increment = 1` が指定されたものとみなす。明示的にこれらを指定した例を以下に示しておこう。

```
>>> for i in range(0, 5, 2):
...     print('i      = ', i)
...     print('i + 1 = ', i+1)
i      = 0
i + 1 = 1
i      = 2
i + 1 = 3
i      = 4
i + 1 = 5
```

このように Python の `for` ループは C のそれと非常に似通っており、また Fortran の `do` ループとも細かい違いを除けば同じように使うことが出来るので、まずはこの使い方が理解できていれば問題ない^{*1}。

当然ながら、以下のように入れ子にすることで多重ループにすることも可能である。

```
>>> for i in range(1, 3):
...     for j in range(1, 5):
...         print('i = ', i, ', j = ', j, ', i*j = ', i*j)
i = 1 , j = 1 , i*j = 1
i = 1 , j = 2 , i*j = 2
i = 1 , j = 3 , i*j = 3
i = 1 , j = 4 , i*j = 4
i = 2 , j = 1 , i*j = 2
```

(次のページに続く)

^{*1} 巷で出くわす Python のループは例えば

```
l = ['a', 'b', 'c']

for index, name in enumerate(l):
    print(index, name)
```

のように複雑怪奇なものが多いことに気がつくであろう。これはイテレータなるものを使って反復処理をもう少し一般的にしたものである。これは少しばかり高度な話題なのでまずは置いておくことにしよう。

(前のページからの続き)

```

9 i = 2 , j = 2 , i*j = 4
10 i = 2 , j = 3 , i*j = 6
11 i = 2 , j = 4 , i*j = 8

```

もう少し実用的な例として、テイラー展開の公式を用いて $\sin(x)$ の近似値を求める例を以下に示す。

```

>>> import math
>>>
>>> x = 0.2
>>>
>>> print('*** Taylor expansion of sin(x) ***')
>>>
>>> y = x
>>> a = x
>>> i = 1
>>> print('x = ', x)
>>> print('i = ', i, ' --- sin(x) = ', y)
>>>
>>> for i in range(3, 10, 2):
...     a = -a / ((i-1)*i) * x**2
...     y = y + a
...     print('i = ', i, ' --- sin(x) = ', y)
>>>
>>> print('exact    --- sin(x) = ', math.sin(x))
*** Taylor expansion of sin(x) ***
x = 0.2
i = 1 --- sin(x) = 0.2
i = 3 --- sin(x) = 0.19866666666666669
i = 5 --- sin(x) = 0.19866933333333336
i = 7 --- sin(x) = 0.19866933079365082
i = 9 --- sin(x) = 0.19866933079506174
exact --- sin(x) = 0.19866933079506122

```

4.3 反復処理 (while)

while を使って for の場合と同様に 0, 1, 2, 3, 4 を出力するループは以下のように書くことができる。

```

>>> i = 0
>>> while i < 5:
...     print(i)
...     i += 1
0
1
2
3
4

```

ただし `i += 1` は `i = i + 1` と同じ、また `i -= 1` であれば `i = i - 1` と同じ意味である。これらの文法は C と同じであるが Fortran にはない書き方である。(ただしインクリメント演算子 `++` やデクリメント

演算子 `--` は Python には存在しない.)

この例では `for` と基本的に同じ動作をするものであるが, `while` はループの終了判定に任意の条件式を用いることができるため, あらかじめ回数の決まっていないループの実行など, より細かい制御を行うことができる. 例えば変数 `x` の値が十分小さくなるまで反復計算をするには

```
>>> x = 1.0
>>> while abs(x) > 1.0e-10:
...     print('x is not small enough')
...     # x がゼロに近づくまで何かの計算をする
```

のような処理を行なうことになる. この例も C なら

```
double x = 1.0;
while( abs(x) > 1.0e-10 ) {
    // 何かの計算
}
```

Fortran なら

```
real(8) :: x = 1.0
do while( x > 1.0e-10 )
    ! 何かの計算
end do
```

と同様である. 以下はテイラー展開の公式を持ちいて, $\sin(x)$ の値が組み込み関数で求められる値 (真の値) に十分近づくまで反復計算をする例である.

```
>>> import math
>>>
>>> x = 0.2
>>> i = 1
>>> y = x
>>> a = x
>>> ytrue = math.sin(x)
>>> print('i = ', i, ' --- sin(x) = ', y)
>>>
>>> while abs((ytrue - y)/ytrue) > 1.0e-10:
...     i += 2
...     a = -a / ((i-1)*i) * x**2
...     y = y + a
...     print('i = ', i, ' --- sin(x) = ', y)
>>>
>>> print('approximated = ', y)
>>> print('exact          = ', ytrue)
>>> print('rel. error    = ', abs((ytrue-y)/ytrue))
i = 1 --- sin(x) = 0.2
i = 3 --- sin(x) = 0.19866666666666669
i = 5 --- sin(x) = 0.19866933333333336
i = 7 --- sin(x) = 0.19866933079365082
approximated = 0.19866933079365082
```

(次のページに続く)

(前のページからの続き)

```

24 exact      = 0.19866933079506122
25 rel. error = 7.0992315183418576e-12

```

このような場合は繰り返しの回数が事前には分からないので、for を使うよりも while を使う方が素直に処理を記述することができる。

4.4 複雑な反復処理

for, while のいずれも continue や break で、より細かいループの制御を行なうことができる^{*2}。使い方は C のそれと同じである。Fortran では continue は cycle, break は exit と同じ意味である。

以下の例は condition1 が満たされたときに「処理 1」を、condition2 が満たされたときに「処理 2」を、どちらも満たされなかったときには「処理 3」を実行する。

```

for i in range(10):
    if condition1:
        # 処理 1
        break
    elif condition2:
        # 処理 2
        continue
    # 処理 3

```

以下の例は while ループで continue や break を使う例である。

```

>>> i = 1
>>> while True:
...     i += 1
...     if i%2 == 0:
...         print('Multiple of 2 --- ', i)
...         continue
...     elif i%3 == 0:
...         print('Multiple of 3 --- ', i)
...         continue
...     elif i%5 == 0:
...         print('Multiple of 5 --- ', i)
...         continue
...     elif i >= 10:
...         print('Exit')
...         break
...     print('Not a multiple of 2, 3, 5 --- ', i)
Multiple of 2 --- 2
Multiple of 3 --- 3
Multiple of 2 --- 4
Multiple of 5 --- 5
Multiple of 2 --- 6

```

(次のページに続く)

^{*2} else 節という構文もあるが、それほど重要ではないのでここでは省略する。

(前のページからの続き)

```

22 Not a multiple of 2, 3, 5 --- 7
23 Multiple of 2 --- 8
24 Multiple of 3 --- 9
25 Multiple of 2 --- 10
26 Exit

```

4.5 関数

複雑でまとまった処理を関数としてあらかじめ定義しておくことで、それを様々な場所から呼び出して使うことができ便利である。関数の詳細には触れずに、まずは最も基本的な使い方だけを身につけよう。

4.5.1 簡単な例

まずは以下の例を見てみよう。

```

>>> # 関数定義
>>> def square(x):
...     return x**2
>>>
>>> # 関数呼び出し
>>> square(2.0)
4.0

```

ここで 2-3 行目が関数の定義である。ここでは与えられた引数の 2 乗を計算し、その値を返す `square` という関数を定義している。6 行目がこの関数の呼び出しであり、引数の 2 乗を計算した値が返ってきたのが分かる。

4.5.2 定義

Python における関数定義は以下のように `def` を用いて行えばよい。

```

def 関数名 (引数 1, 引数 2, ...):
    関数内の処理 (インデントに注意)

```

ここでもインデントによって関数定義ブロックとそれ以外を区別していることに注意しよう。 `def` の次の行のインデントが終わるまでが関数定義とみなされる。先ほどの例 `square` では引数は一つであったが、複数の場合はカンマ , で区切って並べればよい。これらが関数への入力となる^{*3}。

関数の定義位置には注意が必要である。Python はコンパイル型の言語ではないため、上から順に実行され、実行時に各行が評価される。したがって、**関数定義は呼び出しよりも必ず前にされていなければならない**のである。ただし、これはあくまで実行時の順序であってソースコード上での位置 (行数) を意味するものではない。以下の例を見てみよう。

^{*3} 引数を入力だけでなく出力として用いることも可能である。しかしこれには少し複雑な事情があるので、ここでは最も基本的な使い方である引数を入力とし返値を出力とする使い方をまずは覚えておこう。

```

>>> # 以下の呼び出しはエラー (この時点では hello は定義されていない)
>>> #hello1()
>>>
>>> def hello2(name):
...     hello1() # 実行時に hello1 が定義済みであれば OK
...     print('I am', name)
>>>
>>> # 以下の呼び出しもエラー (この時点では hello2 が呼び出す hello1 は定義されていない)
>>> #hello2('John')
>>>
>>> def hello1():
...     print('Hello')
>>>
>>> hello1()
>>> hello2('John')
Hello
Hello
I am John

```

コメントアウトされている 2 行目の `hello1`、9 行目の `hello2` の呼び出しはどちらもエラーとなる。2 行目は、この時点で `hello1` が定義されていないので明らかである。一方で 9 行目については、この時点で `hello2` が定義されているので一見問題ないように見えるが、`hello2` の内部 (5 行目) で呼び出される `hello1` がまだ定義されていないのでエラーになるのである。これは Jupyter Notebook を常用しているとセルの評価順序がバラバラになってしまうので曖昧になりやすい点なので注意しておこう。

4.5.3 返値

関数から呼び出し元に制御を戻すときには `return` を用いる。 `return` には関数の返値を指定することもできるが、省略することもできる。このときはデフォルトで `None` を返す^{*4}。なお、 `return` がなくても関数定義ブロックの最後まで到達したときには呼び出し元に制御が戻り、 `None` が返値となる。Fortran では関数とサブルーチンが区別されているが、Python ではサブルーチンは存在せず、 `None` を返す関数が Fortran のサブルーチンに相当すると考えておけばよいだろう。

4.5.4 変数のスコープ

関数の中では自由に変数を定義して使うことができる。例えば以下の例をみてみよう。

```

>>> import math
>>>
>>> # approximation of exp(x) via taylor expansion up to order n
>>> def exp_taylor(x, n):
...     c = 1.0
...     f = 1.0
...     for i in range(n):
...         c /= (i+1)

```

(次のページに続く)

^{*4} これは「何でもない」ということを示す Python に組み込みのオブジェクトである

(前のページからの続き)

```

9      ...      f += c * x**(i+1)
10     ...      return f
11
12 >>> print('--- exp_taylor ---')
13 >>> x = 0.5
14 >>> print('x      = ', x)
15 >>> print('1st    = ', exp_taylor(x, 1))
16 >>> print('2nd    = ', exp_taylor(x, 2))
17 >>> print('3rd    = ', exp_taylor(x, 3))
18 >>> print('4th    = ', exp_taylor(x, 4))
19 >>> print('5th    = ', exp_taylor(x, 5))
20 >>> print('6th    = ', exp_taylor(x, 6))
21 >>> print('7th    = ', exp_taylor(x, 7))
22 >>> print('8th    = ', exp_taylor(x, 8))
23 >>> print('exact = ', math.exp(x))
24 --- exp_taylor ---
25 x      = 0.5
26 1st    = 1.5
27 2nd    = 1.625
28 3rd    = 1.6458333333333333
29 4th    = 1.6484375
30 5th    = 1.6486979166666667
31 6th    = 1.6487196180555554
32 7th    = 1.6487211681547618
33 8th    = 1.6487212650359622
34 exact = 1.6487212707001282

```

この関数 `exp_taylor` はテイラー展開の公式を n 次まで用いて $\exp(x)$ の近似値を求める。ここで5行目や6行目のように変数を定義して用いている。これらの変数はローカル変数と呼ばれ、この関数の内部でのみ有効な変数である。すなわちこの関数の外で `c` や `f` という変数は(これとは別に定義しない限り)定義されない。もし関数の外で同じ名前の変数を定義して用いたとしても、それらはこの関数内の変数とは全く無関係である。

一方で、関数の外で定義された変数に関数内からアクセスしたいこともあるかもしれない。例えば、以下の関数 `fibonacci` は呼ばれるたびにフィボナッチ数列を返す関数である。

```

>>> # global variables for fibonacci
>>> a = -1
>>> b = 1
>>>
>>> def fibonacci():
...     # a and b refer to the global variables
...     global a, b
...     c = a + b
...     a = b
...     b = c
...     return c
>>>
>>> print('--- fibonacci ---')
>>> for i in range(20):

```

(次のページに続く)

(前のページからの続き)

```
15     ...     print(fibonacci())
16     --- fibonacci ---
17     0
18     1
19     1
20     2
21     3
22     5
23     8
24     13
25     21
26     34
27     55
28     89
29     144
30     233
31     377
32     610
33     987
34     1597
35     2584
36     4181
```

ここでは `a`、`b` という変数は関数の外 (トップレベル) で定義された変数であり、これらに以前の数列の値を保持している。このような変数をグローバル変数と呼ぶ。関数 `fibonacci` からこれらの変数にアクセスするには 22 行目のように `global` を用いて、グローバル変数を用いることを明示する。この例では `a` および `b` は関数内で値が更新されているが、これによってグローバル変数の値も更新される。

このように関数内では基本的に関数内部でのみ有効なローカル変数を用い、グローバル変数へのアクセスが必要なときのみ、それを `global` によって明示すればよい^{*5}。ただし、グローバル変数は気をつけて使わないとバグの原因となりやすいため推奨できない。引数を使って明示的に値を渡すか、場合によってはクラスを用いて実装する方がよいことが多い。

4.6 オブジェクト

ここまではあえて曖昧にしてきたが、実は Python では全て (変数、関数、モジュールなど) がオブジェクト (正確には組み込みの `object` クラスのインスタンス) である。このことを正確に理解するにはオブジェクト指向の考え方が必要になるので、ここでは詳細には立ち入らない。Python はオブジェクト指向についての知識がなくても「なんとなく」使えるという意味で簡単な言語であるが、本格的に使うには、オブジェクト指向やクラス、インスタンス、イテレータなどの少々難解な概念を理解する必要があるが出てくる。そういう意味では Python を本当に使いこなすのは少し難しいかもしれない。Python のいいところは初心者でもある程度簡単に使えるのに加えて、上級者には強力な機能を提供しているところとも言えるかもしれない。ここでは Python を使うにあたって必要最低限の知識だけを身につけておこう。

^{*5} 厳密に言うともう少し複雑な振る舞いをするのだが、グローバル変数を使う際には明示的に `global` を付けておいた方が可読性の観点からよいので、このような記述にとどめた。

4.6.1 属性

一般にオブジェクトは属性 (attribute) と呼ばれるオブジェクトに紐付けられた変数を持つ。例えば「人間」には「名前」という属性があるし、「学生」には「学生証番号」という属性がある、といった具合にそれぞれのオブジェクトの実体が固有の変数を持っていると考えればよい。

オブジェクトに紐付けられた属性にアクセスするには `.` を用いる。例えば `sys` というモジュールもオブジェクトなので、いくつかの属性を持っている。以下の例を見てみよう。

```
>>> import sys
>>> sys.version
'3.7.10 (default, Feb 26 2021, 18:47:35) \n[GCC 7.3.0]'
```

ここでは `sys.version` で `sys` オブジェクトの `version` という属性にアクセスしている。属性は普通の変数と全く同じように用いることができる。

4.6.2 メソッド

属性が変数であったのに対して、メソッド (method) はオブジェクトに紐付けられた関数と考えればよい。例えば「人間」には「走る」という動作があり、「学生」には「学校に行く」という動作がある。メソッドはこのようなオブジェクトの動きを表すものと考えるとよい。ただし、必ずしもオブジェクトの動きを表すようなメソッドでなくてもよく、単にオブジェクトに紐付けられた関数と考えることも多い。

例えば Python プログラムを終了するには `sys.exit()` 関数を使うことは [Python の基本](#) で見た通りであるが、これは `sys` オブジェクトの `exit()` メソッドを呼び出していると考えればよい。他の例も見てみよう。

```
>>> s = 'python'
>>> s.upper()
'PYTHON'
>>> s.lower()
'python'
>>> s.capitalize()
'Python'
```

この例では文字列 `s` のメソッド `upper()`、`lower()`、`capitalize()` をそれぞれ呼び出している。その名前と実行結果から明らかなように、それぞれ大文字に変換、小文字に変換、先頭だけ大文字で以降は小文字に変換した文字列を返す関数である。

4.7 第4章 演習課題

注釈: 以下の課題は Jupyter Notebook の使用を前提としているが、もちろん他の実行環境でも同等の処理を実現出来ていれば問題ない。

参考:

解答例 (ダウンロード)

4.7.1 課題 1

サンプルを実行して動作を確認しよう.

4.7.2 課題 2

2つの整数 m および n を引数として受け取り, その大小を比較する関数 `compare`

```
def compare(m, n):  
    "m と n を比較する"  
    pass
```

を作成しよう. これを呼び出すと

```
>>> compare(2, 1)  
2 is larger than 1  
>>> compare(1, 2)  
1 is smaller than 2  
>>> compare(3, 3)  
3 is equal to 3
```

などと出力するようにすること.

4.7.3 課題 3

西暦 (整数) を引数として受け取り, うるう年かどうか判定する関数 `is_leapyear`

```
def is_leapyear(year):  
    "うるう年の判定"  
    pass
```

を作成しよう. 引数がうるう年であれば真 (`True`), そうでなければ偽 (`False`) を返すものとする. ただしうるう年の判定条件は以下の通りである.

- 400 で割り切れる年は無条件でうるう年である.
- 400 で割り切れずに 100 で割り切れる年はうるう年ではない.
- 100 で割り切れずに 4 で割り切れる年はうるう年である.

以下はこの関数の呼び出しの例である.

```
>>> is_leapyear(2000)  
True
```

4.7.4 課題 4

整数 N を引数として受け取り, $0^\circ \leq \theta \leq 180^\circ$ を N 分割した点 (端点を含むので $N+1$ 点), およびそれらの点における $\sin \theta$, $\cos \theta$ の値を出力する関数

```
def trigonometric(n):
    " $\theta$ ,  $\sin \theta$ ,  $\cos \theta$  の値を出力する"
    pass
```

を作成せよ. これを呼び出すと例えば以下のような出力が得られるものとする.

```
>>> trigonometric(18)
+0.000e+00 +0.000e+00 +1.000e+00
+1.000e+01 +1.736e-01 +9.848e-01
+2.000e+01 +3.420e-01 +9.397e-01
+3.000e+01 +5.000e-01 +8.660e-01
+4.000e+01 +6.428e-01 +7.660e-01
+5.000e+01 +7.660e-01 +6.428e-01
+6.000e+01 +8.660e-01 +5.000e-01
+7.000e+01 +9.397e-01 +3.420e-01
+8.000e+01 +9.848e-01 +1.736e-01
+9.000e+01 +1.000e+00 +6.123e-17
+1.000e+02 +9.848e-01 -1.736e-01
+1.100e+02 +9.397e-01 -3.420e-01
+1.200e+02 +8.660e-01 -5.000e-01
+1.300e+02 +7.660e-01 -6.428e-01
+1.400e+02 +6.428e-01 -7.660e-01
+1.500e+02 +5.000e-01 -8.660e-01
+1.600e+02 +3.420e-01 -9.397e-01
+1.700e+02 +1.736e-01 -9.848e-01
+1.800e+02 +1.225e-16 -1.000e+00
```

ここで三角関数 (`math.sin` および `math.cos`) の引数はラジアン単位であることに注意しよう. なお, 上の例では出力のフォーマットは

```
# x は  $\theta$ , y1, y2 は  $\sin \theta$ ,  $\cos \theta$  の値
print('{: +12.3e} {: +12.3e} {: +12.3e}'.format(x, y1, y2))
```

としている.

4.7.5 課題 5

引数として与えられた 2 つの整数 m および n の最大公約数を求める関数 `gcd`

```
def gcd(m, n):
    " $m$  と  $n$  の最大公倍数を求める"
    pass
```

を作成せよ. これを呼び出すと例えば以下のような出力が得られるものとする.

```
>>> gcd(12, 20)
4
```

なお、最大公約数を求めるには以下のアルゴリズム (ユークリッドの互除法) を用いるとよい。(以下では $m > n$ を仮定していることに注意しよう)

1. m を n で割った余り r を求める.
2. もし $r = 0$ ならば n が最大公約数である.
3. もし $r \neq 0$ ならば, m に n を, n に r を代入して [1] に戻る (繰り返す).

4.7.6 課題 6

以下の級数展開

$$e \simeq \sum_{n=0}^N \frac{1}{n!}. \quad (0! = 1 \text{ に注意せよ})$$

により自然対数の底 e の近似値を求める関数 `approx_e`

```
def approx_e(n, epsilon):
    "自然対数の底を級数展開を用いて求める"
    pass
```

を作成しよう。ただし整数 N および実数 ϵ を引数として受け取り、関数の返値としては、収束ステータス (収束すれば `True` そうでなければ `False`)、反復回数、最終的な近似値の 3 つを返すこと。

ただし以下の条件を満たすこと。

- $N > 1$ でない, または $0 < \epsilon < 1$ でない場合にはエラーメッセージを表示して、収束ステータス、反復回数、近似値の全てを `None` とすること。
- 誤差が ϵ 以下になった時点か、 $n = N$ まで計算した時点で級数計算を打ち切る。

以下はこの関数を呼び出し、結果を表示する例である。

```
>>> status, iteration, e = approx_e(10, 1.0e-8)
>>>
>>> if status:
>>>     print('Converged !')
>>> else:
>>>     print('Did not converge !')
>>>
>>> print('{:20} : {:>20}'.format('N', iteration))
>>> print('{:20} : {:>20.14e}'.format('Approximated', e))
>>> print('{:20} : {:>20.14e}'.format('Exact', math.e))
>>> print('{:20} : {:>20.14e}'.format('Error', abs(e-math.e)/math.e))
Did not converge !
N                               :                10
Approximated                    : 2.71828180114638e+00
Exact                          : 2.71828182845905e+00
Error                           : 1.00477663102111e-08
```

4.7.7 課題 7

与えられた実数 $a(> 0)$ の平方根の近似値を以下のような逐次近似

$$x_{n+1} = \frac{1}{2} \left(\frac{a}{x_n} + x_n \right)$$

で計算して返す関数 `approx_sqrt`

```
def approx_sqrt(a, epsilon):  
    "a の平方根の近似値を求める"  
    pass
```

を作成しよう。ここで `epsilon` は許容誤差である。

なお、 x_n は \sqrt{a} の n 番目の近似値である。初期値としては $x_0 = a$ を与え、 $\|x_{n+1} - x_n\| < \epsilon \|x_n\|$ を満たすまで反復を繰り返せばよい。

以下はこの関数を呼び出し、結果を表示する例である。

```
>>> sqrtal = approx_sqrt(2.0, 1.0e-5)  
>>> sqrtal2 = math.sqrt(2.0)  
>>>  
>>> print('{:20} : {:>20.14e}'.format('Approximated', sqrtal))  
>>> print('{:20} : {:>20.14e}'.format('Exact', sqrtal2))  
Approximated          : 1.41421356237469e+00  
Exact                  : 1.41421356237310e+00
```

第 5 章

便利な組み込み型

Python の強みは言語そのものよりも、その強力なライブラリ群にあると言える。ここでは Python 本体に組み込まれているオブジェクト (データ型) の中でも最も頻繁にお目にかかるであろうもののうちいくつかを簡単に紹介する。ここで紹介するのはあくまで基本中の基本なので、本格的に使うにはこれだけでは少々心もとないが、これをマスターしておけばあとは各自で調べることができるであろう。

参考:

[sample.ipynb](#) (ダウンロード)

この章の内容

- 便利な組み込み型
 - *tuple*
 - *list*
 - *range*
 - *dict*
 - 第 5 章 演習課題

5.1 tuple

tuple と呼ばれるオブジェクトは配列のように複数の値をひとまとめにしたものである。後述の list と同様に、とりあえずは C や Fortran の配列と同じようなものとして考えてもらって構わない。(ただし、通常はあまりサイズの大きなものには使われない。)

tuple は複数の値を () と , (カンマ) で以下のように表される

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

これは 3 つの整数要素から成る tuple を表す。

配列とは異なり、tuple の要素として任意の Python オブジェクトを格納できる。また () は空の (要素が何もない) tuple を表す。なお、tuple の各要素は [] で参照することができる。これらを以下の例で確かめてみよう。

```
>>> () # 空の tuple
()
>>> a = ('this', 'is', 'tuple') # tuple オブジェクトを変数 a に格納
>>> (1, 'string', a) # 異なる型のオブジェクトも格納できる
(1, 'string', ('this', 'is', 'tuple'))
>>> a[0] # a の最初の要素を参照
'this'
```

tuple の大きな特徴は **要素の値を変更することができない** という点である。要素の値を書き換えようとすると以下のようなエラーとなる。

```
>>> a[0] = 'that'
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-28-900cc7775085> in <module>
----> 1 a[0] = 'that'

TypeError: 'tuple' object does not support item assignment
```

tuple は複数の値を一時的にまとめて扱いたいときによく使われるものなので、その要素を書き換えることは想定されていないのである。典型的な使い方の一つとして、関数から複数の値を返したいときなどに使われる。例えば以下の test_tuple は与えられた 2 つの引数の和と差を返す関数である。

```
>>> def test_tuple(a, b):
...     return a+b, a-b
...
>>> test_tuple(1, 2)
(3, -1)
>>> c, d = test_tuple(20, 10)
>>> c
30
>>> d
10
```

この例のように、実は tuple を作成するときの () は省略することができる。ここでは return a+b, a-b でサイズ 2 の tuple (a+b, a-b) を関数の返値としている。また、関数の返値として tuple が返されるときは、その要素をそれぞれ別の変数に代入して使うことができる (この例では c, d)。なお、この例のように tuple から一旦要素を取り出してしまえば、それらの値は当然変更も可能である。

5.2 list

基本的には list は要素の書き換えができる tuple と考えてよい。tuple の例をそのまま list に置き換えてみよう。

```
>>> []
[]
>>> a = ['this', 'is', 'list']
>>> [1, 'string', a]
[1, 'string', ['this', 'is', 'list']]
```

(次のページに続く)

(前のページからの続き)

```

6 >>> a[0]
7 'this'
8 >>> a[0] = 'that' # 自由に要素の書き換えができる
9 >>> a
10 ['that', 'is', 'list']

```

list は C や Fortran の配列と異なり各種メソッドを呼び出すことで値の挿入，追加，削除などが自在にできる。使い方は以下の例を見れば明らかだろう。

```

>>> a = [1, 2, 3]           # 初期化
>>> a.append(4)             # 最後に新しい要素を追加
>>> a
[1, 2, 3, 4]
>>> a.insert(1, 100)        # 1 番目に要素 100 を追加
>>> a
[1, 100, 2, 3, 4]
>>> a.remove(100)          # 要素 100 を削除 (複数ある場合は最初に見つかった要素)
>>> a
[1, 2, 3, 4]
>>> a.pop()                # 最後の要素削除
4
>>> a
[1, 2, 3]
>>> a.extend([10, 11, 12])  # 引数に受け取ったリストの各要素を追加
>>> a
[1, 2, 3, 10, 11, 12]
>>> a.clear()              # 空にする
>>> a
[]

```

注意すべきは list のサイズを伸ばす `append()`, `extend()` を多用すると一般的には遅くなる可能性が高いということである。基本的には大規模なデータの格納には list は使わない^{*1}，また何らかの理由で使わざるを得ないときにはサイズが予め分かっているのであれば最初にそのサイズの list を作っておく (サイズを変更しない)，という方針にすべきである。

また，以下のように tuple を list に変換することもできる。

```

>>> list((3, 4, 5)) # tuple (3, 4, 5) を list に変換
[3, 4, 5]

```

なお，任意のサイズの初期化された list は簡単に作ることができる。例えば以下の例は 0 で初期化されたサイズ 10 の list を作る例である。ループで `append()` を繰り返して作るという無駄なことは絶対にやめよう。

```

>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

```

list を生成するときに，いわゆる「内包表記」を使うと便利なことも多い。例えば以下のように用いる。ただし，慣れるまでは難しいかもしれないので無理して使う必要はない。

^{*1} ほとんどの場合は NumPy 配列を使えば十分なハズである。 [NumPy 配列の基本](#) 参照。

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

5.3 range

これまでも for ループで繰り返し回数を指定するときに `for i in range(10)` のような書き方をしてきた。range は整数列を表すという点を除けば、list や tuple と同じように値の列を表すオブジェクトである。(従って tuple, list, range を総称してシーケンス型と呼ぶ。)

range は規則的な数列を表すオブジェクトであるため、実際にはメモリ上に値を保持せず、必要に応じてその場で値を計算する。すなわち、tuple や list と異なり、常に使用するメモリは一定であるという特徴がある。(したがって for ループなどでの使用に向いている。)

必要であれば、以下のように list や tuple に簡単に変換することが可能である。

```
>>> list(range(5))  
[0, 1, 2, 3, 4]  
>>> tuple(range(5))  
(0, 1, 2, 3, 4)
```

5.4 dict

dict は辞書 (dictionary) や連想配列 (ハッシュ) などと呼ばれるオブジェクトである。複数の値を保持するという意味では list などのオブジェクトと同様である。大きな違いは、list を始めとするシーケンス型は要素が順番を持ち、整数添字で各要素を指定するのに対して、dict の要素は順番を持たない。その代わりに文字列の「キー」と対応する「値」のペアで要素を記憶するのが dict オブジェクトである。dict は多くの値を保持しなければいけないときにも配列やリストのように「順番」を気にせず使うことができるので大変便利である。

以下で簡単な使い方を見てみよう。まずはキーと値のペアから dict オブジェクトを作成する。

```
>>> a = {'key1' : 'value1', 'key2' : 'value2', 'key3' : 'value3'}  
>>> a  
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}  
>>> b = dict(key1='value1', key2='value2', key3='value3')  
>>> b  
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

この例ではどちらも中身は同じもので、'key1', 'key2', 'key3' がキー、'value1', 'value2', 'value3' がそれぞれのキーに対応する値である。なお、値を一意に指定するために dict のキーはユニークである必要がある。そのためキーの重複は許されず、仮に重複があった場合には一番最後に指定された値で上書きされることになるので注意しよう。

具体的に要素にアクセスしたり、追加、削除は以下のように行えばよい。


```
>>> a['key3']          # キーを指定して値にアクセス
'value3'
>>> a['key1'] = 'hoge'  # キーを指定して値を更新
>>> a
{'key1': 'hoge', 'key2': 'value2', 'key3': 'value3'}
>>> a['key0'] = 'value0' # 新たなキーを指定して値を追加
>>> a
{'key1': 'hoge', 'key2': 'value2', 'key3': 'value3', 'key0': 'value0'}
>>> del a['key1']       # 削除
>>> a
{'key2': 'value2', 'key3': 'value3', 'key0': 'value0'}
```

なお、存在しないキーを指定して値にアクセスするとエラーとなる。

```
>>> a['key1']
-----
KeyError                                Traceback (most recent call last)
<ipython-input-26-fbbb4b4f3c5e> in <module>
----> 1 a['key1']

KeyError: 'key1'
```

実際にはエラーとならないように処理が必要である。キーが存在するかどうかは `in` 演算子でチェックすることができるし、キーが存在しない要素にアクセスした時に「デフォルト値」を返すような処理も有効である。以下で例を見てみよう。

```
>>> 'key1' in a
False
>>> 'key2' in a
True
>>> a.get('key1', 'hoge')
'hoge'
>>> a.get('key2', 'hoge')
'value2'
```

1 行目と 3 行目はそれぞれキーがあるかどうかをチェックしている。5 行目および 7 行目は `get()` メソッドを使ってキーに対応する値にアクセスしているが、このとき 2 番目の引数にデフォルト値を指定することができる (指定しない場合は `None`)。 `get()` でアクセスした場合には `[]` とは異なり、キーが見つからなかったときにエラーを出さずに、デフォルト値を返す。これによりエラーチェック処理を省いたり簡単にすることができる。

`dict` 全体を順番に操作する処理には `keys()`、`values()`、`items()` といったメソッドを用いることになる。それぞれキー、値、キーと値のペアの「メモリビュー」オブジェクトを返す。

```
>>> a.keys()
dict_keys(['key2', 'key3', 'key0'])
>>> a.values()
dict_values(['value2', 'value3', 'value0'])
>>> a.items()
dict_items([('key2', 'value2'), ('key3', 'value3'), ('key0', 'value0')])
```

ここでは「メモリビュー」が何かといった細かいことは置いておいて、具体的な使い方の例を見てみよう。これらは以下のようにループで用いることになるだろう。

```
# 全てのキーと値のペアを出力
>>> for key, val in a.items():
...     print("key = {:5s} : value = {:5s}".format(key, val))
key = key2   : value = value2
key = key3   : value = value3
key = key0   : value = value0
# 全てのキーを出力
>>> for key in a.keys():
...     print(key)
key2
key3
key0
```

例えばキーをソートしたいことがあるかもしれない。dict_keys オブジェクトは直接ソートすることができないが、以下のように list に一度変換してしまえば簡単にソートができる。

```
>>> keys = list(a.keys()) # dict_keys を list に変換
>>> keys
['key2', 'key3', 'key0']
>>> keys.sort()           # list オブジェクトなのでソートできる
>>> keys
['key0', 'key2', 'key3']
>>> for key in keys:
...     print("key = {:5s} : value = {:5s}".format(key, a[key]))
key = key0   : value = value0
key = key2   : value = value2
key = key3   : value = value3
```

5.5 第 5 章 演習課題

注釈: 以下の課題は Jupyter Notebook の使用を前提としているが、もちろん他の実行環境でも同等の処理を実現出来ていれば問題ない。

参考:

解答例 (ダウンロード)

5.5.1 課題 1

サンプルを実行して動作を確認しよう。

5.5.2 課題 2

tuple の便利な使い方として関数の引数の扱いが挙げられる。

```
>>> def f(*args):  
...     print(args) # 全ての引数を出力
```

のように引数に * を指定すると、それ以降に与えた引数は全て tuple として引数 (この場合は args) に格納される。これは多数のあらかじめ個数の分からない引数を受け取るために使われる。これは位置指定引数 (positional argument) などと呼ばれる。

任意の個数・任意の型の引数を受け取り、受け取った全ての引数を出力する print_args 関数を作成せよ。例えば以下のような結果が得られればよい。Python の任意のオブジェクトを文字列表現に変換する str 関数を用いること。

```
>>> print_args('hello', (1, 2, 3), [], {'key': 'val'}, None)  
args[ 0] = hello  
args[ 1] = (1, 2, 3)  
args[ 2] = []  
args[ 3] = {'key': 'val'}  
args[ 4] = None
```

なお、あらかじめ定義されている tuple を関数に渡すときに * を使って

```
>>> a = (0, 1, 2)  
>>> print_args(*a)
```

のように渡すこともできる。この使い方も覚えておくと便利である。

5.5.3 課題 3

任意の list を受け取りそこから重複するものを除いた新たな list オブジェクトを返す関数 unique を作成せよ。ただし重複するもの以外の順番は保持するものとする。例えば以下のような結果が得られればよい。(やり方は色々あるが素直にループで処理する方法を考えてみよう。)

```
>>> unique(['a', 1, 'b', 2, 'c', 1, 2, 3, 'b', 'd', 'a', 3])  
['a', 1, 'b', 2, 'c', 3, 'd']
```

5.5.4 課題 4

dict は関数のキーワード引数を受け取るときに用いると便利である。

```
>>> def f(**kwargs):
...     print(kwargs)
```

のように引数に ** を指定すると、それ以降の任意のキーワード引数が dict として引数 (この場合 kwargs) に格納される。

任意個数のキーワード引数の引数名 (キー) とその値を出力する関数 process_kwargs を作成せよ。ただし、引数 a, b は必ず与えられるものとし、もし与えられなかった場合にはデフォルトで None を出力するものとせよ。また引数名で辞書順にソートして出力せよ。

```
>>> process_kwargs(a='hi', c='chao', x=10, y=20, z=30)
a hi
b None
c chao
x 10
y 20
z 30
```

なお、あらかじめ定義されている dict を関数に渡すときに ** を使って

```
>>> a = {'a' : 'hi', 'c' : 'chao', 'x' : 10, 'y' : 20, 'z' : 30}
>>> print_kwargs(**a)
```

のように渡すこともできる。さらに、位置指定引数と合わせて

```
>>> def f(*args, **kwargs):
...     print(args)
...     print(kwargs)
```

のように使うことも可能である。注意すべきは必ず **位置指定引数の方がキーワード引数より先になければならない** という点である。

5.5.5 課題 5

JSON(JavaScript Object Notation) は特にウェブを介したデータのやり取りによく使われるデータの記述形式である。例えば以下は [JSON GENERATOR](#) というサイトで作ったダミーの JSON データの抜粋である。このように JSON は Python の list と dict を組み合わせたような形式で書かれている。

```
[
  {
    "age": 28,
    "name": {
      "last": "Morales",
      "first": "Hickman"
```

(次のページに続く)

(前のページからの続き)

```
7      },
8      "email": "hickman.morales@electonic.cg",
9      "index": 0,
10     "company": "Electonic",
11     "favoriteFruit": "apple"
12   },
13   {
14     "age": 59,
15     "name": {
16       "last": "Norris",
17       "first": "Emerson"
18     },
19     "email": "emerson.norris@recriSYS.sj",
20     "index": 1,
21     "company": "RecriSYS",
22     "favoriteFruit": "strawberry"
23   },
24   {
25     "age": 47,
26     "name": {
27       "last": "Garza",
28       "first": "Shelly"
29     },
30     "email": "shelly.garza@signity.mw",
31     "index": 2,
32     "company": "Signity",
33     "favoriteFruit": "orange"
34   },
35   {
36     "age": 30,
37     "name": {
38       "last": "Owens",
39       "first": "Rhoda"
40     },
41     "email": "rhoda.owens@kidgrease.cn",
42     "index": 3,
43     "company": "Kidgrease",
44     "favoriteFruit": "strawberry"
45   },
46   {
47     "age": 18,
48     "name": {
49       "last": "Dodson",
50       "first": "Florence"
51     },
52     "email": "florence.dodson@musanpoly.cy",
53     "index": 4,
54     "company": "Musanpoly",
55     "favoriteFruit": "banana"
56   }
57 ]
```

このような JSON 形式のデータは Python 標準の `json` モジュールを用いることで非常に簡単に扱うことができる。ここでは実際にこの JSON 文字列データから `first name` と `last name` および `email` を取り出して表示する関数 `process_json1` を作成せよ。この例の JSON 文字列をコピーアンドペーストで `json_string` という変数に格納し、実行すると以下のような結果が得られる。

```
>>> process_json1(json_string)
Hickman MORALES <hickman.morales@electonic.cg>
Emerson NORRIS <emerson.norris@recrisys.sj>
Shelly GARZA <shelly.garza@signity.mw>
Rhoda OWENS <rhoda.owens@kidgrease.cn>
Florence DODSON <florence.dodson@musanpoly.cy>
```

注釈: `json.loads(json_string)` によって JSON 文字列を `list` に変換することができる。 `list` の各要素が `dict` オブジェクトになっているので、それらを逐一処理していけばよい。

5.5.6 課題 6

課題 5 で扱った JSON 文字列から個人を特定できる情報を削除した JSON 文字列を返す関数 `process_json2` を作成せよ。例えば以下のような結果が得られればよい。

```
>>> print(process_json2(json_string))
[
  {
    "age": 28,
    "index": 0,
    "company": "Electonic",
    "favoriteFruit": "apple"
  },
  {
    "age": 59,
    "index": 1,
    "company": "Recrisys",
    "favoriteFruit": "strawberry"
  },
  {
    "age": 47,
    "index": 2,
    "company": "Signity",
    "favoriteFruit": "orange"
  },
  {
    "age": 30,
    "index": 3,
    "company": "Kidgrease",
    "favoriteFruit": "strawberry"
  },
  {
```

(次のページに続く)

(前のページからの続き)

```
28     "age": 18,  
29     "index": 4,  
30     "company": "Musanpoly",  
31     "favoriteFruit": "banana"  
32 }  
33 ]
```

注釈: `json.dumps(object, indent=2)` によって与えられたオブジェクトを JSON 文字列に変換することができる。ここで引数 `indent` に与える数値は文字列にしたときのインデント数である。

第 6 章

NumPy 配列の基本

NumPy と呼ばれるライブラリの存在は Python が広く科学技術計算に用いられるようになった最も大きな要因の一つであろう。ここでは科学技術計算で用いられる大規模データの効率的な扱いを可能にする NumPy の多次元配列オブジェクトについて学ぼう。なお、以降はこの多次元配列オブジェクトを便宜的に NumPy 配列と呼ぶことにする。

参考:

[sample.ipynb](#) (ダウンロード)

この章の内容

- NumPy 配列の基本
 - NumPy について
 - 基本的な使い方
 - 配列オブジェクト
 - 生成
 - 入出力
 - 各種演算
 - インデックス操作
 - 形状操作
 - 第 6 章 演習課題

6.1 NumPy について

NumPy は Python による科学技術計算の根幹をなすライブラリである^{*1}。このライブラリが提供する `numpy` というモジュールを使うためには

```
>>> import numpy as np
```

^{*1} これは Python に組み込みのライブラリではないので別途インストールする必要があるが、ほとんどの環境には入っていると思われるのでここではあるものとして考える。

のようにするのが一般的である。これは `numpy` という名前のモジュールを `import` し、`np` という別名を付けることを意味する。別名は必ずしもつけなくてもいいし、その別名を `np` にする必然性は何もない。しかし、世の中のほとんどのコードがこの書き方を採用しているので、ここでも慣例に従っておこう。以降では常に `np` が `numpy` モジュールを意味するものとする。

NumPy 配列 (NumPy が提供する多次元配列オブジェクト) は正確には `np.ndarray` オブジェクトである^{*2}。一般に Python はスクリプト言語に分類される動作の遅い言語であるが、NumPy 配列を用いることで非常に高速な数値演算が可能となる。これは NumPy 配列が C で実装されており、NumPy 配列の演算は「うまく」使えばほとんどが内部的には C のコードとして実行されるからである。詳細は以下で追々見ていくことにしよう。

6.2 基本的な使い方

まずは基礎の基礎を抑えよう。例えば配列を作る最も簡単な方法の一つとして

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

のように `np.arange()` を使う方法がある。`np.arange()` の引数に配列の長さを指定することで連番の整数配列ができる。この例では 10 が指定されているので、長さ 10 の整数配列が生成されている。

次に $0 \leq x < 2\pi$ の範囲を等間隔に N 分割し、その各点で $\cos(x)$ を計算してみよう。

```
>>> N = 10
>>> x = np.arange(N) / N * 2 * np.pi
>>> y = np.cos(x)
```

ここで整数の変数 N が配列のサイズである。2 行目で配列 x を生成し、3 行目でその配列の各要素について $\cos(x)$ の値を計算している。

ここで 2 行目では `np.arange()` で生成した配列に対してスカラー値の掛け算や割り算 (整数 / 整数 = 実数に注意) を実行していることが分かるだろう (`np.pi` は π の値である)。このとき、配列の各要素に対して演算がそれぞれ実行される。同様に 3 行目では配列の各要素に対して `np.cos()` 関数を呼び出し、結果として生成される新しい配列を y に代入している。このような使い方は **Fortran の配列演算の使い方と全く同じ** である。C や Python の組み込みの配列^{*4} では当然このような書き方は許されないが、NumPy 配列はあたかも Fortran の配列のように使えるのである。

このようにループを使わずに配列演算を記述できると確かに非常に便利であるが、NumPy 配列の本当の利点は単に簡便さではなく、その演算速度にある。例えば `y = np.cos(x)` を (長さ N の配列 x および y が定義されているとして) Python のループを使って書くならば、

```
>>> for i in range(N):
>>>     y[i] = np.cos(x[i])
```

^{*2} 正確にはクラス `np.ndarray` のインスタンスと呼ぶべきかもしれないが、ここでは細かいことは置いておこう。

^{*4} Python にも配列を提供する `array` という組み込みのモジュールがあるのだが、NumPy 配列があまりにも便利で普及しているのでそれほどお目にかかることはないだろう。

のようになるだろう^{*3}。ここで NumPy 配列の各要素へのアクセスには C のように [] を使って行うことに注意しよう。計算自体は同じことだが、このループは Python で実行されるため、その演算速度は配列演算を使った場合に比べて桁違いに遅い。これは配列演算は実質的には C で書かれた NumPy ライブラリ内で実行されることになるためである。すなわち、Python 上で実行される処理は遅いのだが、Python から呼び出した関数が C や Fortran で実装されていれば、その部分については C や Fortran で高速に処理することができるのである。

Python は様々なことが手軽に実現できる非常に便利な言語であるが、ネックとなるのはその処理の遅さである。(したがって、何も考えずに書くといとも簡単に遅いコードが出来上がってしまう。) NumPy 配列を使うことで重たい処理だけを他の言語で書かれた高速なライブラリに押し付けることができ、手軽さと計算速度の速さを両立しているわけである。したがって、**Python ではループ処理をするのは可能な限り避け、可能な限り配列演算を利用するのが鉄則**である。

6.3 配列オブジェクト

NumPy 配列オブジェクト `np.ndarray` は多次元配列データを格納し効率的な演算を実現する。

6.3.1 属性

NumPy 配列の場合は当然ながら各要素をデータとして持つが、それに加えて以下の属性が重要である。

属性	意味
<code>dtype</code>	配列要素のデータ型 (整数, 実数, 複素数など)
<code>ndim</code>	次元数
<code>size</code>	配列サイズ (要素数)
<code>shape</code>	配列形状

以下の例を見てみよう。 `np.arange()` で生成された配列 `x` について見てみると、これは 64bit の整数 (`int64`) 配列であり、次元数は 1、は配列サイズは 10 である。最後の形状は各次元の長さを tuple で表したものであるが、この場合は 1 次元配列なので `(10,)` となっている。例えば 3 次元配列で各次元の長さが 3, 4, 5 だったとすると `(3, 4, 5,)` がその配列の形状となる。

```
>>> x = np.arange(10)
>>> x.dtype
dtype('int64')
>>> x.ndim
1
>>> x.size
10
>>> x.shape
(10,)
```

(次のページに続く)

^{*3} このように NumPy の数学関数はスカラーに対しても NumPy 配列に対しても有効である。さらに、標準の `math` モジュールよりも多くの数学関数が用意されているので、特に理由がない限りは常に NumPy の数学関数を用いるのが簡単であろう。

```

10 >>> np.array([[1, 2, 3], [4, 5, 6]]).shape
11 (2, 3)

```

6.3.2 データ型

NumPy 配列に格納されるデータ型 (`.dtype` 属性) はほとんどの場合は C や Fortran の組み込み型と同等のものである。以下に代表的な NumPy のデータ型とそれに対応する C や Fortran のデータ型を示す。

NumPy	型コード	C	Fortran	備考
<code>np.int32</code>	<code>'i4'</code>	<code>int32_t</code>	<code>integer(4)</code> / <code>integer</code>	32 ビット整数 (通常は C の <code>int</code> , NumPy の <code>'i'</code>)
<code>np.int64</code>	<code>'i8'</code>	<code>int64_t</code>	<code>integer(8)</code>	64 ビット整数 (通常は C の <code>long</code> , NumPy の <code>'l'</code>)
<code>np.uint32</code>	<code>'u4'</code>	<code>uint32_t</code>		32 ビット符号なし整数 (通常は C の <code>unsigned int</code>)
<code>np.uint64</code>	<code>'u8'</code>	<code>uint64_t</code>		64 ビット符号なし整数 (通常は C の <code>unsigned long</code>)
<code>np.float32</code>	<code>'f4'</code>	<code>float</code>	<code>real(4)</code> / <code>real</code>	32 ビット浮動小数点数 (通常は NumPy の <code>'f'</code>)
<code>np.float64</code>	<code>'f8'</code>	<code>double</code>	<code>real(8)</code>	64 ビット浮動小数点数 (通常は NumPy の <code>'d'</code> , Python 組み込みの <code>float</code>)
<code>np.complex64</code>	<code>'c8'</code>	<code>float complex</code>	<code>complex(4)</code> / <code>complex</code>	実部・虚部が共に 32 ビット浮動小数点数の複素数
<code>np.complex128</code>	<code>'c16'</code>	<code>double complex</code>	<code>complex(8)</code>	実部・虚部が共に 64 ビット浮動小数点数の複素数 (Python 組み込みの <code>complex</code>)

少し古いデータ型の指定方法では一文字の型コードが使われていた。例えば

```

>>> np.dtype('d')
dtype('float64')
>>> np.dtype('i')
dtype('int32')

```

の例では `'d'` は `np.float64`, `i` は `np.int32` に対応している。しかし、これは環境に依存して変わってしまう可能性がある (例えば `'i'` が 64 ビット整数 `np.int64` の環境もおおくはない) ので、NumPy では明示的にデータサイズを示した `np.int32` のような指定が推奨されている。

6.4 生成

配列を生成するにはいくつかの方法があるが、以下に何も無いところから配列を作るための典型的な例をいくつか示そう。

6.4.1 シーケンス型からの生成

既存の list や tuple を `np.array()` 関数の引数に与えることで NumPy 配列に変換することができる。使い方は以下の例を見れば明らかであろう。

```
>>> np.array([0, 1, 2, 3, 4])           # list [0, 1, 2, 3, 4] を変換
array([0, 1, 2, 3, 4])
>>> np.array((0, 1, 2))                # tuple (0, 1, 2) を変換
array([0, 1, 2])
>>> np.array([[0, 1], [2, 3], [4, 5]])  # list の list を 2 次元配列に変換
array([[0, 1],
       [2, 3],
       [4, 5]])
```

6.4.2 arange

すでに見たように `np.arange()` は連番の配列を作る。いくつか別の使い方も見てみよう。

```
>>> np.arange(10, 20, 2)
array([10, 12, 14, 16, 18])
```

のように生成される配列の始点、終点、ステップを指定することもできる。指定の仕方は `for` ループでよく用いられる `range()` と同様である。また、

```
>>> np.arange(10, dtype=np.float64)
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

のように `dtype` を指定することで、生成される配列のデータ型を指定することもできる。

6.4.3 zeros

`np.zeros()` は 0 で初期化された配列を作るための関数である。以下の例

```
>>> np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

ではサイズが 10 の実数配列を生成している。デフォルトのデータ型が倍精度実数 (`np.float64`) であることに注意しよう。このデータ型は `dtype` オプションで指定することができる。

多次元の配列を生成するには、以下のように形状を指定すればよい。

```
>>> np.zeros((2, 4), dtype=np.int32)
array([[0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int32)
```

この例では形状として (2, 4) を指定しているので 0 で初期化された 2 次元配列ができている。

さらに `np.zeros_like()` を使うと、既に存在する別の配列と同じ形状・同じデータ型の配列を生成することもできる。

```
>>> x = np.zeros((3, 3), dtype=np.float32)
>>> np.zeros_like(x)
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]], dtype=float32)
```

6.4.4 ones

`np.ones()` は 1 で初期化する以外は `np.zeros()` と同じように使うことができる。例えば以下は 2 次元の配列を生成する。

```
>>> np.ones((3, 3))
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

ここでもデフォルトのデータ型が倍精度実数であることに注意しよう。

同様に `np.ones_like()` も `np.zeros_like()` の代わりに使うことができる。

```
>>> x = np.zeros((3, 3), dtype=np.float32)
>>> np.ones_like(x)
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

この例のように引数で与えられた配列の形状およびデータ型だけを受け継ぎ、その値は 1 で初期化されていることに注意しよう。

6.4.5 linspace

ある区間を均等に区切った点列を配列として作成したいときに便利なのが `np.linspace()` である。例えば $0 \leq x \leq 1$ を均等に区切った配列を生成してみよう。

```
>>> np.linspace(0.0, 1.0, 11)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

ここで `np.linspace()` がデフォルトで端点(この場合は $x = 0$ と $x = 1$)を含むことに注意しよう。 $x = 1$ を含まなくても良いのであれば `np.arange()` を用いて

```
>>> np.arange(10)/10
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

としてもよい。

6.4.6 geomspace / logspace

`np.linspace()` によってある区間 $[a, b]$ を線形に均等に区切ることができた。同様に与えられた区間を対数スケールで等間隔に区切るために `np.geomspace()` と `np.logspace()` が提供されている。この両者の違いは `np.geomspace()` には区間の端点の値をそのまま引数として指定するのに対して、 `np.logspace()` には $\log_{10}(a)$ および $\log_{10}(b)$ を引数として渡すことである。

```
>>> np.geomspace(0.1, 10.0, 11) # [0.1, 10] を log で等間隔に区切る
array([ 0.1      ,  0.15848932,  0.25118864,  0.39810717,  0.63095734,
        1.      ,  1.58489319,  2.51188643,  3.98107171,  6.30957344,
        10.     ])
>>> np.logspace(-1, +1, 11)      # 全く同じ (log10(0.1) = -1, log10(10) = +1)
array([ 0.1      ,  0.15848932,  0.25118864,  0.39810717,  0.63095734,
        1.      ,  1.58489319,  2.51188643,  3.98107171,  6.30957344,
        10.     ])
```

上の2つの例は丸め誤差を除いて以下と全く同一の結果を与える。

```
>>> 10**np.linspace(np.log10(0.1), np.log10(10.0), 11)
array([ 0.1      ,  0.15848932,  0.25118864,  0.39810717,  0.63095734,
        1.      ,  1.58489319,  2.51188643,  3.98107171,  6.30957344,
        10.     ])
```

6.4.7 random

乱数配列を作るのも以下のように簡単にできる。

```
>>> np.random.random(5) # [0, 1) の一様乱数 (長さ 5 の配列)
array([0.80086594, 0.2513905 , 0.41056114, 0.14083936, 0.74182892])
>>> np.random.randint(0, 5, size=10) # [0, 10) の整数乱数 (長さ 10 の配列)
array([2, 4, 4, 3, 1, 0, 4, 2, 2, 1])
>>> np.random.random((3, 2)) # [0, 1) の一様乱数を形状 (3, 2) の 2 次元配列として
array([[0.56945673, 0.5676268 ],
       [0.39896178, 0.0149435 ],
       [0.35165915, 0.93962764]])
```

なお、ここで `np.random.random()` というのは `numpy` モジュールの中の `random` モジュールの中の `random()` 関数という意味である。

6.5 入出力

NumPy は配列の入出力用に便利な関数が用意されており、非常に簡単に使うことができる。

6.5.1 テキスト入出力

loadtxt

テキスト形式のデータファイルを読み込むには `np.loadtxt()` が大変便利である。ここではデータファイル `helix1.dat` を読み込んでみよう。このファイルの中身は以下のようになっており、値がスペースで区切られている。

```
0.81 0.59 0.50
0.31 0.95 1.00
(中略)
0.81 0.59 15.50
0.31 0.95 16.00
```

このファイルは以下のように 1 行で読み込むことができる。

```
>>> x = np.loadtxt('helix1.dat')
>>> x.shape
(32, 3)
```

読み込んだ結果は 2 次元配列として返される。この場合は 32 行 x 3 列のデータのため `.shape` 属性は `(32, 3)` となっている。`np.loadtxt()` には多くのオプションがあるが、重要なものは以下の通りである。

パラメータ名	データ型	概要
<code>dtype</code>	<code>np.dtype</code>	デフォルトでは <code>float (np.float64)</code>
<code>delimiter</code>	<code>str</code>	値の区切り文字 (デフォルトではスペース)
<code>comments</code>	<code>str</code>	各行のこの文字列に続く部分はコメントとみなされ無視する (デフォルトでは <code>'#'</code>)
<code>skiprows</code>	<code>int</code>	ファイル先端から指定された行数分だけ無視する (デフォルトでは 0)

例えばスペースではなくカンマ `,` で区切られたいいわゆる CSV(Comma-Separated Values) ファイルを読み込むには

```
>>> x = np.loadtxt('filename.csv', delimiter=',')
```

のように区切り文字を指定すればよい。

savetxt

`np.loadtxt()` の逆に配列をファイルに出力するには `np.savetxt()` を使えばよい。例えば NumPy 配列オブジェクト `x` を `output.dat` というファイルに出力するには

```
>>> np.savetxt('output.dat', x)
```

とすればよい。代表的なオプションをいくつか紹介しておこう。

パラメータ名	データ型	概要
<code>fmt</code>	<code>str</code>	フォーマット文字列
<code>delimiter</code>	<code>str</code>	値の区切り文字 (デフォルトではスペース)
<code>comments</code>	<code>str</code>	ヘッダー・フッター用のコメント文字 (デフォルトでは '#')
<code>header</code>	<code>str</code>	ファイル先端のコメントヘッダー
<code>footer</code>	<code>str</code>	ファイル終端のコメントフッター

例えば `helix1.dat` を読み込み、同じフォーマットで別のファイル `output.dat` に書き込むには

```
>>> x = np.loadtxt('helix1.dat')
>>> np.savetxt('output.dat', x, fmt='%5.2f')
```

のようにフォーマット文字列を指定すればよい。

6.5.2 バイナリ入出力

NPY 形式と NPZ 形式

単一の NumPy 配列をバイナリのまま (メモリ上のバイト列をそのまま) ファイルに保存するファイル形式が NPY 形式である。これは NumPy の独自フォーマットであるが、その形式は [公開](#) されている。さらに、複数の NumPy 配列を単一のファイルに書き込む形式が NPZ 形式である。まずは NumPy 配列をこれらの形式で保存する例を見てみよう。NPY 形式には `np.save()`、また NPZ 形式には `np.savez()` をそれぞれ用いる。例として、

```
>>> x = np.linspace(0.0, np.pi, 100)
>>> y = np.cos(x)
```

によって作成した配列を保存してみよう。

NPY 形式で保存するには

```
>>> np.save('binary1.npy', x)
```

とすればよい。最初の引数がファイル名、2 番目の引数が保存する配列である。また、NPZ 形式で複数の配列を保存するには

```
>>> np.savez('binary1.npz', x_name=x, y_name=y)
```

のようにすればよい。ここで `np.savez()` には複数の配列を与えることになるため、それぞれに名前(この例では `x_name` および `y_name`)を与えていることに注意しよう。なお、ここではそれぞれ拡張子を `.npz`, `.npz` としたファイル名を与えたが、そうでない場合には自動的にこれらの拡張子が付与される。

どちらの形式でもファイルを読み込むときには `np.load()` を用いればよい。NPY 形式の場合は `np.load()` は読み込んだデータを直接 NumPy 配列として返すので、

```
>>> xx = np.load("binary1.npy")
```

のように使えばよい。ここでは `xx` に読み込まれた配列が代入される。NPZ 形式の場合には `np.load()` は dict-like なオブジェクトを返す。すなわち、

```
>>> data = np.load("binary2.npz")
```

のように読み込むと `data` は dict のように振る舞うオブジェクトが代入される。したがって、

```
>>> xx = data['x_name']
>>> yy = data['y_name']
```

によってそれぞれ NumPy 配列が得られる。ここで `x_name` や `y_name` は `np.savez()` に与えた配列の名前である。

生のバイナリファイル

C 言語の `fwrite()` 関数や Fortran のストリーム出力で作成されたバイナリファイルは他の情報が何も付与されていない「生」のバイナリファイルである。このような生のバイナリファイルは `np.fromfile()` で読み込むことができる。基本的には Python 組み込みの `open()` 関数でファイルを開き、`np.fromfile()` でデータを読めばよい。

```
>>> # 読み込み専用 (r) でファイルを開く
>>> fp = open('cbinary.dat', 'r')
>>> # np.float64 のデータを 10 個読み込み NumPy 配列として z に代入
>>> z = np.fromfile(fp, np.float64, 10)
>>> # ファイルを閉じる
>>> fp.close()
```

生のバイナリファイルには余計な情報は一切付与されていないので、データの形式や個数をあらかじめ知っていなければ正しく読み込むことができないことに注意しよう。逆に言えば、それさえ理解してあれば自由自在にファイルを扱うことができる。基本的にファイルの操作は [公式ドキュメント](#) に従って行えばよく、読み込んだバイト列を NumPy 配列として評価したいときにだけ `np.fromfile()` を使えばよい。

unformatted 形式 (Fortran)

Fortran の `unformatted` 形式は、生のバイナリファイルに近いが、ヘッダとフッタが前後に付与されることが多い*8。ここでは `scipy.io.FortranFile` オブジェクトを用いた読み込み方法を紹介しておこう。

ここでは `Fortran` 演習 で用いた `unformatted` 形式のバイナリファイル `helix2.dat` を読み込んでみよう。`FortranFile` オブジェクトの `read_record()` メソッドは Fortran の 1 回の `write` 文に対応するデータを読み込み、NumPy 配列を返す。このファイルには 3 つの倍精度実数の配列が 3 回の `write` 文で保存されているので、

```
>>> from scipy.io import FortranFile
>>> # ファイルを開く
>>> fp = FortranFile('helix2.dat')
>>> # 3 回の read_record でそれぞれ NumPy 配列として読み込み
>>> x = fp.read_record(np.float64)
>>> y = fp.read_record(np.float64)
>>> z = fp.read_record(np.float64)
>>> # ファイルを閉じる。
>>> fp.close()
```

のように 3 回の `read_record()` メソッド呼び出しでそれぞれ読み込めばよい。倍精度実数データを読み込むので `np.float64` を引数に与えていることに注意しよう。なお、1 行目は `scipy.io` モジュールから `FortranFile` オブジェクトを読み込むことを意味する。

HDF5 と netCDF

バイナリファイルは高速に読み書きができ、またテキストへの変換がないため、情報が失われることが無いという利点がある。その一方で、可搬性に欠けるという欠点があり、ある計算機で生成したデータが他の計算機ではそのままでは読めないという問題が起こることがある。また、データ読み込みの際にはあらかじめ何が、どのような形式で書かれているかが分からなければデータの読み込みができない。このような欠点を補うファイルフォーマットとして比較的広く使われているのが HDF5 である。C 言語や Fortran で HDF5 を扱うのは少々面倒であるが、Python であれば `h5py` というモジュールを使うことで簡単に操作できる。

また同じような目的で `netCDF` というデータ形式もあり、気象などの分野で使われているようである。最近の `netCDF` は実は中身は HDF5 で書かれているようなので、python であればやはり `h5py` で操作が出来るし、`netCDF` 専用の Python モジュール (`netcdf4-python`) も存在する。

数値シミュレーションには C/C++ や Fortran などの高速な言語が用いられることが多いが、シミュレーションの結果をこれらの形式で出力しておけば、Python を用いた結果の解析が非常にやりやすくなるのである。

*8 ただし、`unformatted` 形式は標準化されておらず、コンパイラやエンディアン (CPU の種類) などに依存する形式なので、可能な限りその使用を避けた方が無難である。(比較的新しいストリーム入出力を使えばよい。)

6.6 各種演算

以下では配列に関する各種演算について見てみよう。

6.6.1 算術演算

以下の例を見れば一目瞭然であろう。基本的に Fortran の配列と同様な演算ができると思ってよい。

```
>>> a = np.arange(4)
>>> b = np.arange(4) + 4
>>> a + b
array([ 4,  6,  8, 10])
>>> a - b
array([-4, -4, -4, -4])
>>> a * b
array([ 0,  5, 12, 21])
>>> a / b
array([0.         , 0.2         , 0.33333333, 0.42857143])
>>> a**2
array([0, 1, 4, 9])
```

6.6.2 ユニバーサル関数

NumPy 配列を引数にとり各要素にそれぞれ適用される関数は、NumPy ではユニバーサル関数 (ufunc) と呼ばれている (通常の関数と同じものだと考えてよい)。よく使うであろう数学関数は全て提供されていると考えてよい^{*5}。Fortran などでは提供されていない関数の使用例をいくつか示しておこう。なお、以下の例はスカラーに対して関数を呼び出しているが、配列を引数に与えた場合には各要素に対して演算した結果を返す。

```
>>> np.deg2rad(90.0)      # degree から radian への変換
1.5707963267948966
>>> np.rad2deg(np.pi/2)  # radian から degree への変換
90.0
>>> np.lcm(4, 6)          # 最小公倍数
12
>>> np.gcd(12, 20)        # 最大公約数
4
>>> np.angle(1.0 + 1.0j)  # 複素数の偏角
0.7853981633974483
```

詳細は [公式ドキュメント](#) を参照のこと。

^{*5} NumPy で提供されていないような特殊関数であっても、ほとんどは SciPy の特殊関数モジュール `scipy.special` に実装されているであろう。ここで提供されていない関数が必要になることは稀である。

6.6.3 メソッド

NumPy 配列も当然オブジェクトなのでメソッドを持つ。また、多くのメソッドにはそれと対応する関数が存在する。以下で例を見てみよう。

配列の総和を計算するには `np.sum()` という関数を用いることができる。

```
>>> x = np.arange(10)
>>> np.sum(x)
45
```

これと同じことを配列 `x` のメソッド `sum()` を使って

```
>>> x.sum()
45
```

と表現することもできる。このように関数とメソッドがどちらも用意されているときにはどちらを使っても全く同じである。(メソッドの方が少しタイプ量が少ないし、タブキーでの補完を考えた時にはよいかもしれない。) 以下にもう少し例を示しておこう。

```
>>> x.max()    # 最大値
9
>>> x.min()    # 最小値
0
>>> x.mean()   # 平均値
4.5
```

単なる数値的な演算に限らず多くの便利なメソッドが用意されている。詳細は [公式ドキュメント](#) を参照のこと。

6.7 インデックス操作

NumPy のインデックス操作は基本的に Fortran の配列操作と似ているので Fortran ユーザーには馴染みやすいだろう。

```
>>> x = np.arange(10)
>>> x[2]                # 3 番目の要素
2
>>> x[2:5]              # 3, 4, 5 番目の要素 (結果も NumPy 配列)
array([2, 3, 4])
>>> x[::2] = -1         # 0, 2, 4, 6 番目の要素に-1 を代入
>>> x                   # 確認
array([-1,  1, -1,  3, -1,  5, -1,  7,  8,  9])
>>> x[::-1]            # 順番を反転
array([ 9,  8,  7, -1,  5, -1,  3, -1,  1, -1])
>>> x[-1]              # 最後の要素
9
>>> x[-2]              # 最後から 2 番目の要素
8
```

Fortran との違いは負のインデックスを指定すると後ろから順に数えた要素を返すところである。実はこのようなインデックス操作は Python 組み込みのシーケンス型 (list など) に対して行うこともできる。NumPy 配列はこのインデックス操作をさらに強力なものにする。例えば、以下のようにインデックスを配列で渡すことも可能である。

```
>>> i = np.arange(1, 10, 2) # インデックス配列を作成
>>> i                        # 確認
array([1, 3, 5, 7, 9])
>>> x[i] *= 2                # 2 倍にする
>>> x                        # 確認
array([-1,  2, -1,  6, -1, 10, -1, 14,  8, 18])
>>> x[i-1]                  # インデックスを一つずらす
array([-1, -1, -1, -1,  8])
```

のように使うことができる。(これは差分法に使うときなどに便利である。)

さらに、多次元配列のインデックス操作についても例を見てみよう。

```
>>> y = np.arange(20).reshape((4, 5)) # 長さ 20 の配列を reshape で 2 次元配列に変換
>>> y
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> y[0,:]                    # 1 次元目は 0, 2 次元目は全て
array([0, 1, 2, 3, 4])
>>> y[:,1]                   # 1 次元目は全て, 2 次元目は 1
array([ 1,  6, 11, 16])
>>> y[1:3,1:3]               # 両次元とも 1, 2 番目の要素
array([[ 6,  7],
       [11, 12]])
```

より高次元の配列を扱う際には ... が便利である^{*6}。

```
>>> z = np.arange(18).reshape((2, 3, 3)) # 3 次元配列
>>> z
array([[[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8]],

      [[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]]])
>>> z[...,0]                 # z[:, :, 0] と同じ
array([[ 0,  3,  6],
       [ 9, 12, 15]])
>>> z[1,...,0]               # z[1, :, 0] と同じ
array([ 9, 12, 15])
>>> z[1]                     # z[1, :, :] と同じ
```

(次のページに続く)

^{*6} ... は Ellipsis Python のオブジェクトというものであり、何か省略されていることを意味する。

(前のページからの続き)

```

16 array([[ 9, 10, 11],
17        [12, 13, 14],
18        [15, 16, 17]])

```

他にも色々便利なのだがあまりにも機能が沢山あるので、これ以上は [公式ドキュメント](#) を参照して欲しい。一つだけ配列の代入について注意しておこう。

```

>>> x[:] = 10 # 配列の全要素に 10 を代入
>>> x         # 確認
array([10, 10, 10, 10, 10, 10, 10, 10, 10, 10])
>>> x = 10    # 整数 10 を x に代入
>>> x         # x はスカラー
10

```

このように 1 行目と 4 行目では動作が異なる。Fortran では 4 行目のように記述しても配列の全要素に 10 を代入してくれるが、Python では **新しい変数を作り、そこに整数 10 を代入** する。これは動的型付け言語の Python ならではの動作である。配列の全要素にスカラー値を代入する際にはインデックスに `:` (もしくは `...`) を指定しなければならない。

6.8 形状操作

配列の形状は決して固定のものではない。実はこれまでの例にも既に出てきたが、

```

>>> x = np.arange(10)
>>> y = x.reshape((2, 5))
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

```

のように `reshape()` メソッドによって配列の形状を変更することができる。この例ではまず長さ 10 の 1 次元配列 `x` を作って、それを形状 (2, 5) の 2 次元配列 `y` に変換している。一体何が起きているのだろうか？ 試しに `y` の要素を変更してみよう。

```

>>> y[1,0] = -1
>>> x
array([ 0,  1,  2,  3,  4, -1,  6,  7,  8,  9])
>>> y
array([[ 0,  1,  2,  3,  4],
       [-1,  6,  7,  8,  9]])

```

ここで変更したのはあくまで `y` の一つの要素であるが、それに伴って `x` の要素も変更されていることに注意しよう。すなわち、`x` も `y` もある連続したメモリブロック (この場合は 8 byte x 10 個 = 80 byte) への参照でしかない (つまりデータの実態は同じもの)^{*7}。 `y` に対しては 2 つのインデックスを使って要素を指定するが、

^{*7} 疑り深い人はもっと直接的に確かめてみよう。 `shares_memory()` 関数は与えられた 2 つの NumPy 配列が同じメモリ領域を

これはあくまで人間から見た時の考えやすさのためにそうしてるに過ぎない。(これは何も Python に限ったことでなく、Fortran でも同じことである。C の場合は少し事情が複雑だが、static な配列に限定すればやはり同じである。)

したがって、当然ながら `reshape()` 前後で要素数が同じである必要がある。試しに `x` を形状 `(3, 5)` に変換しようとしても

```
>>> np.arange(10).reshape((3, 5))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 10 into shape (3,5)
```

のようにエラーとなる。

多次元配列の連続したメモリブロックを新たな 1 次元配列として得るために `ravel()` メソッドを使うことができる。似たようなメソッドとして `flatten()` があるが、これらの違いはなんだろうか。

```
>>> a = y.ravel()
>>> b = y.flatten()
>>> a
array([ 0,  1,  2,  3,  4, -1,  6,  7,  8,  9])
>>> b
array([ 0,  1,  2,  3,  4, -1,  6,  7,  8,  9])
```

見た目ではどちらも同じように見える。ここで `y` の要素を再び変更してみよう。

```
>>> y[1,0] = 5
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b
array([ 0,  1,  2,  3,  4, -1,  6,  7,  8,  9])
```

このように `ravel()` で作られた `a` は同じメモリブロックを参照しているのに対して、`flatten()` で作られた `b` は異なる領域を指していることが分かる。すなわち `flatten()` は新たなメモリブロックを確保し、データをコピーする。一方で、`ravel()` は単なる参照でしかない。当然 `ravel()` はデータをコピーする手間がない分高速に動作することに注意しよう。

なお、ここで紹介した形状操作はあくまでも初級編である。NumPy の演算速度を最大限に活かそうと思うと、このような配列形状の変更には思いのほか頻繁に出くわすことになるので少しずつ慣れていこう。

指しているかどうかをチェックする。これを使うと

```
>>> np.shares_memory(x, y)
True
```

のように、確かに `x` と `y` が同じメモリ領域を指していることがわかる。

6.9 第 6 章 演習課題

注釈: 以下の課題は Jupyter Notebook の使用を前提としているが、もちろん他の実行環境でも同等の処理を実現出来ていれば問題ない。ただし、この章の目的は NumPy 配列の使い方の習得なので、以下の課題では可能な限りループを使わず NumPy の機能を活かした実装を目指そう。

なお、以下では

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
```

のように必要なモジュールが import されているものとする。

参考:

解答例 (ダウンロード)

6.9.1 課題 1

サンプルを実行して動作を確認せよ。

6.9.2 課題 2

整数 N を引数として受け取り、 $0^\circ \leq \theta \leq 180^\circ$ を N 分割した点 (端点を含むので $N+1$ 点)、およびそれらの点における $\sin \theta$ 、 $\cos \theta$ の値をそれぞれ配列として返す関数

```
def trigonometric(n):
    "θ, sin θ, cos θ の値を出力する"
    pass
```

を作成せよ。これを呼び出し、例えば以下のようにプロットして結果を確認しよう。

```
>>> x, y1, y2 = trigonometric(18)
>>> plt.plot(x, y1, 'ro')
>>> plt.plot(x, y2, 'bo')
>>> plt.plot(x, np.sin(np.deg2rad(x)), 'r-')
>>> plt.plot(x, np.cos(np.deg2rad(x)), 'b-')
```

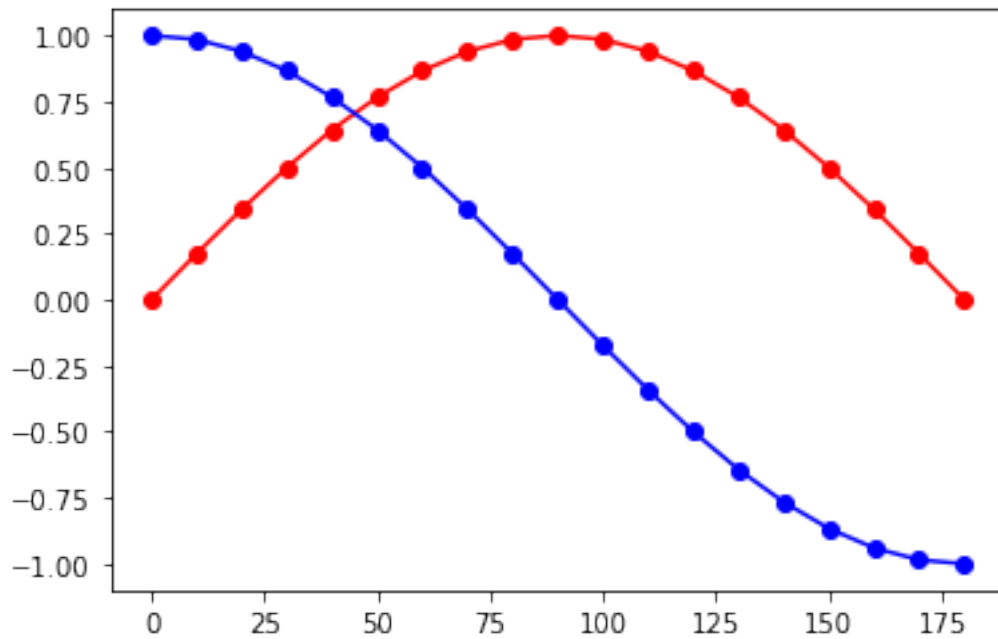


図1 課題2

6.9.3 課題3

以下の級数展開

$$e \simeq \sum_{n=0}^N \frac{1}{n!}. \quad (0! = 1 \text{ に注意せよ})$$

により自然対数の底 e の近似値を求める関数 `approx_e()`

```
def approx_e(n):
    "自然対数の底を級数展開を用いて求める"
    pass
```

を作成しよう。ただし整数 N を引数として受け取り、内部では NumPy 配列を使うことでループを用いない実装とせよ。

ここで、階乗とガンマ関数の関係式

$$\Gamma(n) = (n-1)!$$

を用いてよい。ガンマ関数は `scipy.special.gamma()` を用いればよい。以下はその使用例である。

```
>>> from scipy import special
>>> special.gamma(3) # 2!
2.0
>>> special.gamma(4) # 3!
6.0
```

この関数を用いると例えば以下のような結果が得られる。

```
>>> print('{:20} : {:>20.14e}'.format('Approximated', approx_e(10)))
>>> print('{:20} : {:>20.14e}'.format('Exact', np.e))
Approximated      : 2.71828152557319e+00
Exact             : 2.71828182845905e+00
```

6.9.4 課題 4

以下の漸化式

$$p_{n+1} = p_n + \alpha p_n(1 - p_n)$$

で定義される数列 $p_n (n = 0, 1, \dots)$ を考える. $1 < \alpha < 3$ を m 分割し, 各 α について初期値 $p_0 = 0.9$ から数列を生成し, そのうち $n = 100, \dots, 200$ を α の関数としてプロットしよう.

これには, 整数 m を引数にとり, α と $p_n (n = 100, \dots, 200)$ の組を返す関数 `logistic()`

```
def logistic(m):
    "0 < α < 1 を m 分割して, それぞれに対してロジスティック写像を計算"
    pass
```

を作成し, この関数を用いて

```
>>> a, p = logistic(2000)
>>> plt.scatter(a, p, s=0.001, color='r')
>>> plt.xlim(1.0, 3.0)
>>> plt.ylim(0.0, 1.4)
```

とすればよい. 結果は以下の通りである.

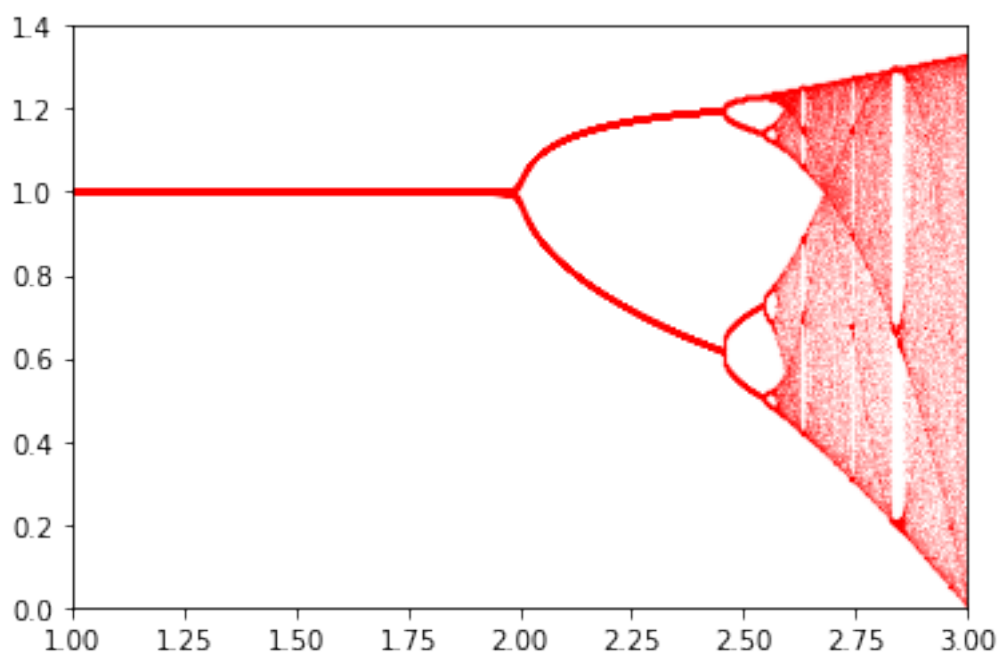


図2 課題4

このような写像はロジスティック写像と呼ばれ、非常に単純な式ながら一定の条件を満たすときにはカオスを生み出すことが知られている。

6.9.5 課題 5

参考:

アルゴリズムについては [こちら](#) を参照.

以下の積分

$$\int_0^1 \frac{4}{\pi} \frac{1}{1+x^2} dx$$

を台形公式および Simpson の公式を使って求める関数をそれぞれ作成しよう. ここで作成する関数は分割数 n を引数にとり,

```
def trapezoid(n):
    "分割数 n の台形公式で関数を積分"
    pass

def simpson(n):
    "分割数 n の Simpson 公式で関数を積分"
    pass
```

のような形になるものとする. ただし関数内ではループを使わずに実装すること. これらの積分公式の誤差の分割数依存性は次のようにすれば調べることができる.

```
>>> M = 16
>>> n = 2**(np.arange(M)+1)
>>> err1 = np.zeros(M, dtype=np.float64)
>>> err2 = np.zeros(M, dtype=np.float64)
>>>
>>> for i in range(M):
>>>     err1[i] = np.abs(trapezoid(n[i]) - 1)
>>>     err2[i] = np.abs(simpson(n[i]) - 1)
>>>
>>> # 結果をプロット
>>> plt.plot(n, err1, 'rx-', label='trapezoid')
>>> plt.plot(n, err2, 'bx-', label='simpson')
>>> plt.loglog()
>>> plt.legend()
```

これによって得られる図は以下の通りである.

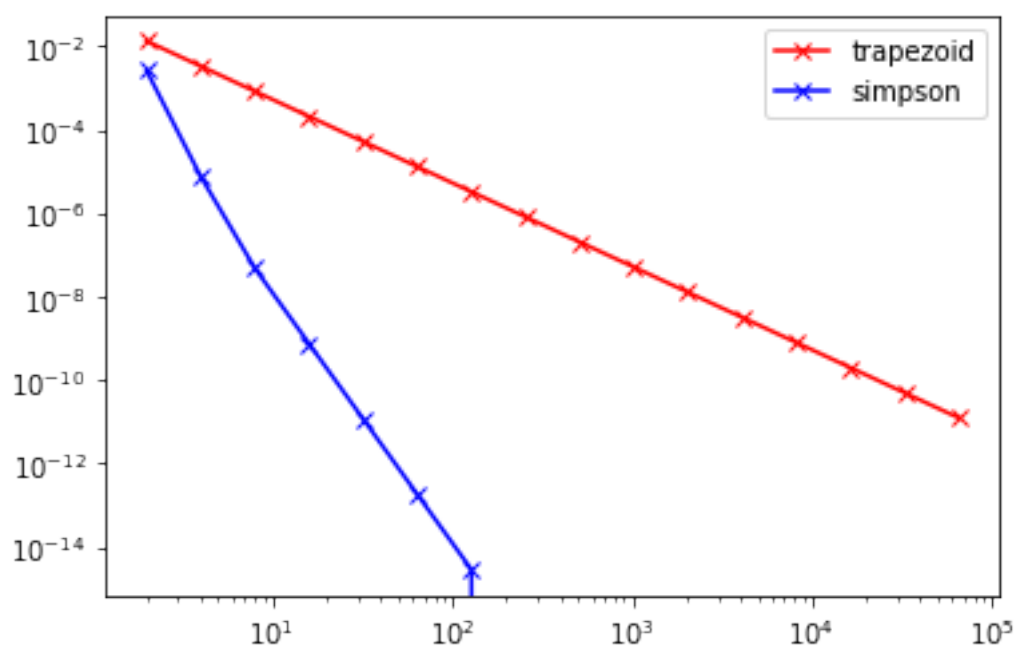


図3 課題5

6.9.6 課題6

データファイルに格納されている学生のテスト点数リストから、最高点、最大点、平均点、標準偏差を求めて返す関数 `analyze_score()` を作成しよう。ただし、

```
def analyze_score(filename):
    "filename からデータを読んで最高点、最低点、平均点、標準偏差を返す"
    pass
```

のようにファイル名を引数にとるものとする。ここで、それぞれ対応する関数 (`np.max()`, `np.min()`, `np.mean()`, `np.std()`) または NumPy 配列のメソッドを用いること。

なお、データファイルは `score1.dat` および `score2.dat` とし、以下のようにこのファイルの最初の行にはデータ数、次の行は空行、以降に実際のデータが格納されている。

```
$ cat score1.dat
30      # データ数
        # 空白行
20      # 実際のデータはこれ以降の行
49
45
(以下略)
```

したがって、`np.loadtxt()` のオプション `skiprows` を用いて最初の2行を無視して読み込めばよい。

例えば `score1.dat` を読み込んだ場合には以下のような結果が得られる。

```
>>> smax, smin, savg, sstd = analyze_score('score1.dat')
```

(次のページに続く)

(前のページからの続き)

```

2 >>>
3 >>> print('{:20} : {:10}'.format('Best', smax))
4 >>> print('{:20} : {:10}'.format('Worst', smin))
5 >>> print('{:20} : {:10.3f}'.format('Average', savg))
6 >>> print('{:20} : {:10.3f}'.format('Standard deviation', sstd))
7 Best                :          98
8 Worst               :           6
9 Average              :       46.400
10 Standard deviation  :       25.115

```

同様に score2.dat についても試してみることに。

6.9.7 課題 7

前問の score1.dat および score2.dat のようなデータファイルを読み込みヒストグラムを作成する関数 hist_score()

```

def hist_score(filename, nbins):
    "filename からデータを読んでヒストグラムを作成する"
    pass

```

を作成しよう。ここで引数 nbins はビンの数である。関数 np.histogram() を用いること。

実行例は以下のようになる。

```

>>> nbins = 20
>>> bins, hist = hist_score('score2.dat', nbins)
>>> binw = bins[+1:] - bins[:-1]
>>> plt.bar(bins[0:-1], hist, width=binw, align='edge')
>>> plt.xlim(0, 100)

```

6.9.8 課題 8

データファイル vector.dat に格納されている 2 つのベクトルを読み込み、それらの内積を求める関数 dot_file()

```

def dot_file(filename):
    "filename から 2 つのベクトルを読み込み、それらの内積を返す"
    pass

```

を作成しよう。関数 np.dot() を用いること。ただし、以下のように vector.dat の最初の行にはベクトルの長さ、次の行は空白、ベクトル 1 の各要素、空白行、ベクトル 2 の各要素、のようにデータが格納されている。

```

20                # データ数
                # 空白行
1.0000000000000000e+00

```

(次のページに続く)

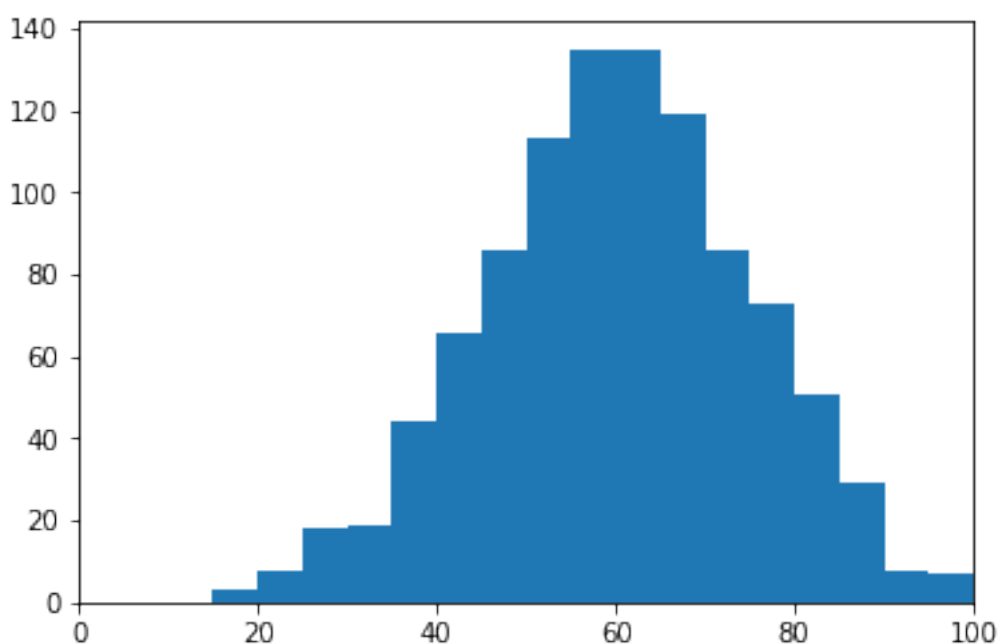


図4 課題7

(前のページからの続き)

```

4 9.510565162951535e-01
5 (中略)
6 8.090169943749473e-01
7 9.510565162951535e-01
8 # 空白行
9 0.0000000000000000e+00
10 3.090169943749474e-01
11 (中略)
12 -5.877852522924734e-01
13 -3.090169943749476e-01

```

`np.loadtxt()` のオプション `skiprows` を用いて最初の2行を無視して読み込むと以降のデータが全て1次元配列として返される(区切りの空白行は無視される)ので、この前半と後半をそれぞれ別のベクトルとして処理すればよい。

6.9.9 課題9

任意の長さの1次元整数型配列に与えられた整数が含まれているかどうかを調べ、含まれていた時には真(True, そうでなければ偽 False)を返す関数 `find_var` を作成しよう。また同様に、値が含まれていた時にはその要素のインデックスを返し、そうでなければ-1を返す関数 `find_var_index()` を作成しよう。実装は色々考えられるが、`np.any()` や `np.nonzero()` または `np.where()` などを使うとよい。

これらの関数を

```

def find_var(x, var):
    "var が配列 x に含まれているかどうかを調べる"

```

(次のページに続く)

(前のページからの続き)

```

3     pass
4
5 def find_var_index(x, var):
6     "var が配列 x に含まれている時にはそのインデックスを、そうでない場合は-1 を返す"
7     pass

```

のように定義し、以下のように動作を確かめよう。

```

>>> x = np.random.randint(0, 100, 10)
>>> x
array([17, 75, 28, 83, 93, 20, 19, 94, 62, 61])
>>> find_var(x, 28)
True
>>> find_var(x, 25)
False
>>> find_var_index(x, 28)
2
>>> find_var_index(x, 25)
-1

```

さらに、`np.random.randint()` を用いて適当な長さの 1 次元整数配列を作成し、値を探索してみよう。

6.9.10 課題 10

参考:

1 次元のセル・オートマトンについては [Wikipedia](#) を参照。

1 次元のセル・オートマトンを実装しよう。具体的には、以下のような関数を作成し、10 進数で与えられたルール `decrule` に基づいて、一次元配列 `x` を初期値とし、`step` 数だけ状態遷移を求めよう。関数の返値は 2 次元配列で、各ステップ、各セルの状態を表すものとする。ただし、境界の値は初期値のまま固定でよい。

```

def cellular_automaton(x, step, decrule):
    "初期値 x から step 数だけセル・オートマトンによる状態遷移を求める"
    pass

```

例えば以下の例ではルール 90 を採用し、初期値として中央のセルのみを 1 とし、それ以外は 0 を与えている。

```

>>> n = 256
>>> m = n//2
>>> xzero = np.zeros((n,), dtype=np.int32)
>>> xzero[m] = 1
>>>
>>> y = cellular_automaton(xzero, m, 90)
>>> plt.imshow(y, origin='lower', cmap=plt.cm.gray_r)

```

これはシェルピンスキーのギャスケット（下図）を生成することが知られている。

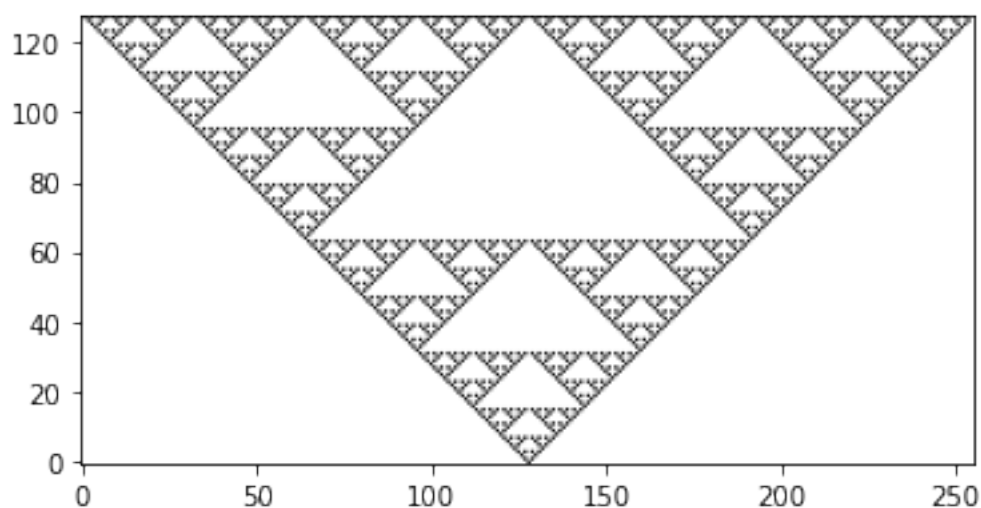


図5 課題 10

6.9.11 課題 11

参考:

アルゴリズムについては [こちら](#) を参照.

逆関数法を使って指数分布

$$P(x) = \lambda \exp(-\lambda x)$$

に従う乱数を返す関数 `randexp()`

```
def randexp(l, n):
    "指数分布に従う乱数を n 個作成する"
    pass
```

を作成しよう。(ただし、実用上は指数分布に従う乱数は `numpy.random.exponential()` によって得られるので、これを用いればよい.)

この関数を用いて発生させた乱数と解析的な確率分布を以下のように比較しよう.

```
>>> l = 1.0
>>> r = randexp(l, 60000)
>>> # ヒストグラム
>>> hist, bins = np.histogram(r, bins=np.linspace(0, 12, 33), density=True)
>>> plt.step(bins[0:-1], hist, where='post')
>>> # 解析的な分布
>>> x = 0.5*(bins[1:] + bins[:-1])
>>> y = l * np.exp(-l*x)
>>> plt.plot(x, y, 'k--')
>>> plt.semilogy()
```

この結果は以下ようになる.

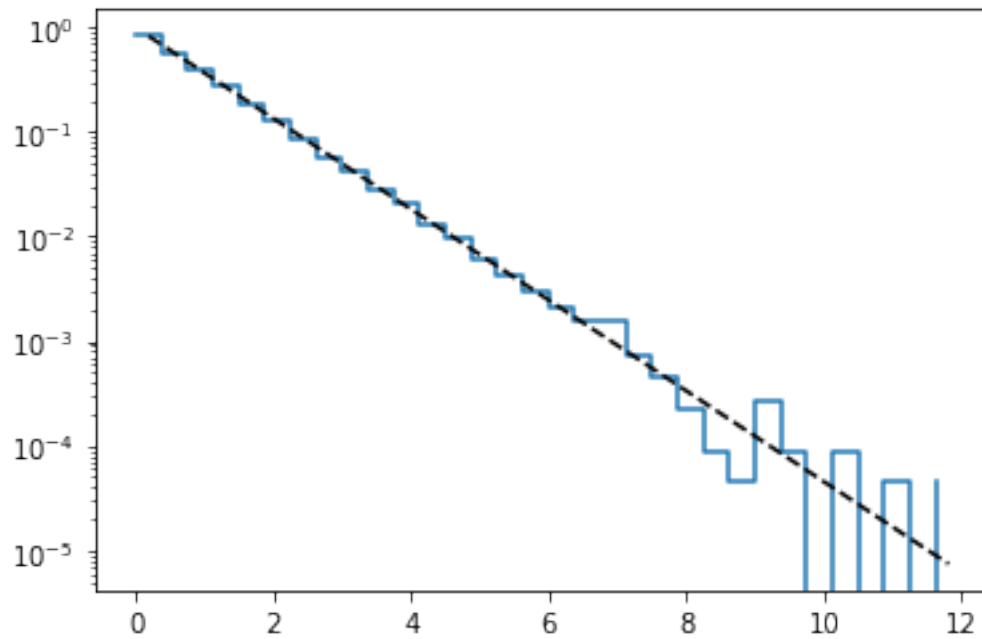


図 6 課題 11

6.9.12 課題 12

参考:

アルゴリズムについては [こちら](#) を参照.

モンテカルロ法を用いて N 次元超球の体積を求める関数 `mc_hypersphere()`

```
def mc_hypersphere(n, nrand):
    "n 次元超球の体積をモンテカルロ法で求める"
    pass
```

を作成しよう. ここで `nrand` は各次元について発生させる乱数の個数である.

この関数を使って近似値と解析的な値を以下のように比較しよう.

```
>>> n = 3
>>> nrand = 100000
>>> print('{:20} : {:<20.14e}'.format('Monte-Carlo', mc_hypersphere(n, nrand)))
>>> print('{:20} : {:<20.14e}'.format('Exact', hypersphere(n)))
Monte-Carlo      : 4.183040000000000e+00
Exact            : 4.18879020478639e+00
```

ただし, 解析的な値を返す関数 `hypersphere()` を別途定義するものとする.

第 7 章

スクリプトとしての Python

これまで見てきたように Python の大きな強みは対話的に実行ができることである。一方で対話的な実行では、毎回必ず人間が何らかの操作をしなければならない。Python でも最低限の人間の操作でまとまった処理を自動化することができる大変便利である。Python では bash などを用いたシェルスクリプトの代替のような使い方から、GUI アプリケーションや、より高度なプログラム開発まで様々なことができる。ここではコマンドラインで実行できる Python スクリプト開発の基本を学ぼう。

参考:

- sample1.py : スクリプトの基本
- sample2.py : ファイルの読み書きと文字列処理
- sample3.py : システムインターフェース
- sample4.py : コマンドライン引数の処理

この章の内容

- スクリプトとしての *Python*
 - スクリプトの基本
 - ファイルの読み書きと文字列処理
 - システムインターフェース
 - コマンドライン引数の処理
 - 第 7 章 演習課題

7.1 スクリプトの基本

7.1.1 実行方法（復習）

ここ で既に見たように、`*.py` という名前で保存した Python のソースコード（スクリプト）はコマンドラインで実行することができる。ここではもう一度 `hello.py` を見直してみよう。

リスト 1 hello.py (再掲)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print('Hello, World !')
```

この一行目の `#!/usr/bin/env python` は Unix-like な OS で

```
$ ./hello.py
```

のように実行するときにこのスクリプトを Python で解釈するためのオマジナイである。最近では諸々の事情でシステム内に Python 環境が複数存在することもあるので、混乱を避けるために以下では

```
$ python hello.py
```

のように実行することにしよう。例えばバージョンの違う複数の Python 環境がシステムに存在するときには陽に自分が使いたい Python のコマンドを指定すればよい。例えば、最近では流石に少なくなったが、古い Python-2.x でしか動かないコードを実行するときには、明示的にそれを指定する必要がある。Python-2.x のパスが `/usr/local/bin/python2` であるときに、これを用いてスクリプトを実行したい場合には

```
$ /usr/local/bin/python2 hello.py
```

のように実行すればよい。

7.1.2 作法

Python のスクリプトでは以下のような書き方がよく見られる。

リスト 2 sample1.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

n = 10
x = 3.14

def main():
    print('main() is called')
    print('n = {}'.format(n))
    print('x = {}'.format(x))

if __name__ == '__main__':
    print('This will be printed only if run as a script')
    main()
```

14 行目以降の `if` ブロックはこのプログラムがスクリプトとして実行された場合にのみ実行される。これは、

このファイルがスクリプトとして実行されたとき（トップレベルスクリプトと呼ぶ）のみ `__name__` という特殊な変数に `__main__` という名前が（Python によって自動的に）与えられるようになっているためである。したがって、このファイルを実行すると以下のような結果が得られる。

```
$ python sample1.py
This will be printed only if run as a script
main() is called
n = 10
x = 3.14
```

11 行目までのコードは変数や関数の宣言のみであり、実行は 14 行目以降で行われていることが分かる。このような書き方がよく用いられるのは、このファイルが他のファイルや、Jupyter Notebook などの対話的環境から import される可能性を考慮しているからである。例えばコマンドラインで以下のように実行してみよう。

```
$ python -c 'import sample1; sample1.main();'
main() is called
n = 10
x = 3.14
```

ここで `-c` オプションで与えられた文字列を Python が解釈して実行している。まず `import sample1` でこのファイルをモジュールとして import し、このモジュールで定義された関数 `sample1.main()` を実行している。この場合は 14 行目以降の `if` ブロックは実行されていないことが分かる。このように単なるスクリプトとしての実行だけでなく、自分が import されたときの場合も考えて

```
if __name__ == '__main__':
    # トップレベルスクリプトとして実行される処理
```

のような形でトップレベルスクリプトとしての実行文とそれ以外を分離しておくのが作法になっている。

注釈: スクリプトを開発している際にも対話型の環境を用いると便利である。例えば、ある機能を実装する関数を実装するときにも、対話型の実行環境でその都度動作確認をしながら実装すると効率が良い。ここで紹介した作法でスクリプトに関数を実装していれば、ソースコードを修正し、その都度 Jupyter Notebook などで `import` や `importlib.reload` しながら関数単体の動作確認ができる（トップレベルスクリプトとしては実行されない）。例えば Jupyter Notebook のセルで

```
>>> import sample1
>>> sample1.main()
```

した後には `sample1.py` を修正したとき、同じ Jupyter セッション中であっても

```
>>> import importlib
>>> importlib.reload(sample1)
>>> sample1.main()
```

のように `importlib.reload` することで、その修正を反映させることができる。`importlib.reload` については [公式ドキュメント](#) を参照のこと。

7.2 ファイルの読み書きと文字列処理

7.2.1 ファイルの読み書き

NumPy 配列の基本 ではすでにバイナリファイルの読み書きについて扱ったが、テキストファイルの扱いもほとんど同様である。例えば以下の例を見てみよう。

```
>>> f = open('testfile.txt', 'w')
>>> f.write('hello')
5
>>> f.close()
```

ここで `open()` には開きたいファイル名および読み書きのモードを与える。この例では書き込み `w` モードでファイルを開く。これによってファイルオブジェクトが返されるので、`write()` メソッドを呼び出してで文字列を書き込み、`close()` で最後にファイルを閉じる。次にこのファイルの中身を読み込んでみよう。これにはモードを `r` としてファイルを開き、`read()` メソッドを呼び出せばよい。

```
>>> f = open('testfile.txt', 'r')
>>> f.read()
'hello'
>>> f.close()
```

ここで `read()` はデフォルトでファイルの中身を全て読み込むが、引数で読み込むデータサイズ（バイト単位）を指定することもできる。また `readline()` は 1 行を読み込み、`readlines()` はファイルの中身を全て読み込み、各行を `list` として返す。

また、一度開いたファイルは必要が無くなったら閉じるのが作法であるが、これを忘れるのを防ぐために最近では `with` が使われることが多い。以下の例を見てみよう。

```
>>> # 書き込み
>>> with open('testfile.txt', 'w') as f:
>>>     f.write('hello\npython')
>>> # 読み込み
>>> with open('testfile.txt', 'r') as f:
>>>     print(f.readlines())
['hello\n', 'python']
```

ここでは `with` で開いたファイルを `f` として、ファイルの読み書きを行っているが `with` のブロックを抜けると、自動的に `close()` が呼ばれるので、閉じ忘れの心配がない。`open()` については [公式ドキュメント](#) も参照しよう。

7.2.2 str

Python による文字列処理には組み込みの文字列オブジェクトである `str` の種々なメソッドを駆使して行えばよい。例えば `split()` は引数で与えられた文字（デフォルトではホワイトスペース）を区切りとして、元の文字列を分割した `list` を返す。また、`join()` は引数で与えられた文字列のシーケンス (`list` や `tuple`) を結合する。使い方は以下の例を見れば明らかであろう。

```
>>> 'I love python'.split()
['I', 'love', 'python']
>>> '_'.join(['I', 'hate', 'fortran'])
'I_hate_fortran'
```

他にも以下の例のように `find()` メソッドを使って文字列を検索したり、配列のように添字 `[]` を用いて部分文字列を取り出したりすることができる。

```
>>> text = 'Earth and Planetary Physics'
>>> text.find('Planet')
10
>>> text[10:]
'Planetary Physics'
```

より詳細については [公式ドキュメント](#) を参照のこと。

7.2.3 re

先の例の `find()` は厳密に一致する文字列を検索するのには十分であるが、より柔軟な文字列の検索・置換には正規表現が用いられる。Python では標準ライブラリの `re` (regular expression) が正規表現を扱うためのモジュールになっている。正規表現そのものの詳細についてはここでは立ち入らないが、ツールによって微妙に正規表現のシンタックス（文法）が異なることがあるので注意しよう。Python の正規表現については [公式ドキュメント](#) を参照して欲しい。

ここでは `re` モジュールの使い方を簡単に見てみよう。

```
>>> url1 = 'https://www.eps.s.u-tokyo.ac.jp'
>>> url2 = 'https://www.eps.s.u-tokyo.ac.jp/epp/'
>>> https_pattern = r'https://([a-zA-Z0-9\-\.\.]+)/?.*'
>>> # 検索
>>> re.search(https_pattern, url1)
<re.Match object; span=(0, 31), match='https://www.eps.s.u-tokyo.ac.jp'>
>>> re.search(https_pattern, url2)
<re.Match object; span=(0, 36), match='https://www.eps.s.u-tokyo.ac.jp/epp/'>
>>> # グループを取り出す
>>> re.search(https_pattern, url1).groups()
('www.eps.s.u-tokyo.ac.jp',)
>>> re.search(https_pattern, url2).groups()
('www.eps.s.u-tokyo.ac.jp',)
```

上のコードでは `re.saerch()` を使って文字列 `url1` および `url2` から `https` プロトコルの URL を表すパターン `https_pattern` を検索している。パターンが見つかったら `re.search()` は `re.Match` オブジェク

トを返す。ここで謎の呪文のようにも見える `https_pattern` が正規表現である。ここで `[a-zA-Z0-9\-\.\.]+` がアルファベット（大文字・小文字）、数字、`-` および `.` の 1 回以上の繰り返しを意味している。このパターンが `()` で囲まれているが、これによって囲まれた範囲がグループ化され、`re.Match` オブジェクトの `groups()` メソッドを呼ぶことで各グループにアクセスすることができる。`re.Match` そのものは URL 全体にマッチしているが、この場合はグループの最初の要素が URL のドメイン部分だけを示している。

以下の例では `re.sub()` を用いて URL の `https` を `http` に置換している。

```
>>> re.sub(r'https://([a-zA-Z0-9\-\.\.]+)/?.*', r'http://\1', url1)
>>> 'http://www.eps.s.u-tokyo.ac.jp'
>>> re.sub(r'https://([a-zA-Z0-9\-\.\.]+)/?.*', r'http://\1', url2)
>>> 'http://www.eps.s.u-tokyo.ac.jp/epp/'
```

ここで `re.sub()` の第 2 引数の `\1` には第 1 引数で指定したパターンの `()` で囲まれたグループが代入される。複数のグループがある場合には `\1`, `\2`, ... のように各グループを指定すればよい。

`re.finditer()` は検索したパターンを順に処理するループを記述するのに便利である。`sample2.py` は Python のソースコードから関数定義を検索するサンプルで、以下には `re.finditer()` を用いたループを抜粋している。第 1 引数 `pattern` にマッチする部分を第 2 引数 `text` から検索して、見つかった順にループ内で処理をしている。

リスト 3 sample2.py（抜粋）

```
for m in re.finditer(pattern, text, re.MULTILINE):
    groups = m.groups()
    if len(groups) >= 1:
        # 関数名
        name = groups[0]
        # 引数リスト
        args = groups[1].split(',')
        while args.count('') > 0:
            args.remove('')
        nargs = len(args)
        print('found function named {} with {} argument(s)'.format(name, nargs))
```

このように `re` モジュールを使うことで柔軟な検索や置換処理が実装できる。

7.3 システムインターフェース

7.3.1 ファイルやディレクトリの操作

ファイルやディレクトリの操作をするための便利な関数群が Python の標準ライブラリで提供されている。`bash` などのシェルスクリプトは基本的には Unix-like な OS での動作が前提となるが、Python のライブラリをうまく使うことで OS などの環境に依らないプログラムを作ることが可能である^{*1}。

これには標準ライブラリの `os` モジュール（およびそのサブモジュール）を `import` して使えばよい。例えば `os.listdir()` は与えられたパス内の全てのファイルとディレクトリを `list` オブジェクトとして返す。した

^{*1} もちろん色々な環境で動くように注意してプログラムを作れば、の話である。

がって、シェルで

```
$ ls
test.txt  testdir/
```

となる環境であれば、

```
>>> os.listdir('.')
['testdir', 'test.txt']
```

なる結果が得られる。この結果として返される list の各要素がファイルかディレクトリか調べるには `os.path.isfile()` や `os.path.isdir()` を使えばよい。例えば

```
>>> for f in os.listdir('.'):
>>>     if os.path.isfile(f):
>>>         print("{} is a file".format(f))
>>>     if os.path.isdir(f):
>>>         print("{} is a directory".format(f))
"testdir" is a directory
"test.txt" is a file
```

のような結果が得られる。また、以下は指定されたディレクトリのファイルのうち Python のソースコードだけを抽出し、各ファイルの行数を数える関数 `count_py_lines1()` の実装例である。

リスト 4 sample3.py (抜粋)

```
def count_py_lines1(dirname):
    print('*** count_by_lines1')
    # ファイルとディレクトリのリスト
    files = os.listdir(dirname)
    files.sort()
    for f in files:
        # 拡張子をチェックして python のソースであれば行数を数える
        if os.path.splitext(f)[1] == '.py':
            with open(f, 'r') as fp:
                c = len(fp.readlines())
                print('{} : number of lines = {}'.format(f, c))
```

ファイルやディレクトリの操作で注意しなければならないのはパスの区切り文字である。Windows ではパスの区切り文字として `\` が使われているのに対して Mac や Linux などでは `/` が使われているので、どちらの環境でも動くスクリプトにするためにはプログラム側でこの違いを吸収してやらなければならない。Python では `os.sep` (または `os.path.sep`) がパスの区切り文字として定義されているので、これを使うことでプラットフォームに依存しないプログラムとすることができる。例えば `os.getcwd()` で現在の作業ディレクトリを取得し、これにファイル名を文字列として結合して絶対パスとするには `os.path.join()` を用いて

```
>>> os.path.join(os.getcwd(), 'test.txt')
/home/hoge/test.txt
```

のようにすればよい。ここでは `os.getcwd()` で得られる `/home/hoge` と `test.txt` が区切り文字 `/` で結合されているが、同じコードを Windows で実行すれば自動的に区切り文字として `\` が選ばれる。同じこ

とが

```
>>> os.sep.join([os.getcwd(), 'test.txt'])
/home/hoge/test.txt
```

でも可能である。これは `str` オブジェクトである `os.sep` の `join` メソッドを使った例である。

他にもファイルの作成やファイル名変更、削除などの一通りの作業ができる。詳細については [公式ドキュメント](#) を確認して欲しい。ただし、ファイルのコピーや削除などのより高レベルの（より簡単な）ファイル操作には標準ライブラリの `shutil` モジュール（[公式ドキュメント](#)）を用いる方が便利であろう。

7.3.2 シェルコマンドの実行

`os` モジュールの `os.system()` を使うと Python プログラムからシェルコマンド（例えば `ls` や `cat` など）を実行することができる。使い方は以下の通りである。

```
>>> r = os.system('echo "hello shell"')
hello shell
>>> print(r)
0
```

ここでシェルで実行されたコマンドの終了ステータスが返値となる。

単にコマンドを実行するだけであればこれでも十分だが、実用的にはシェルコマンドに何らかの入力を与えたり、また実行結果を受け取って処理をしたくなってくる。これには `subprocess` モジュールを使うのがよい。例えば、先ほどの例と同じことは `subprocess.run()` を使って

```
>>> r = subprocess.run('echo "hello shell"', shell=True)
hello shell
>>> print(r.returncode)
0
```

のように実現できる。ここで `subprocess.run()` に `shell=True` としているのは第 1 引数で与えた文字列をシェルで解釈するという意味である。このオプションを指定しないときには

```
>>> r = subprocess.run(['echo', 'hello shell'])
hello shell
```

のように `list` としてコマンドラインに与える引数を与える必要がある。（シェルとして解釈させるのはセキュリティ的にあまり好ましくないので注意する必要がある。）

シェルのパイプ（`|`）のように標準出力を Python で受け取って処理をするには `capture_output=True` と `text=True` を指定すればよい^{*2}。

^{*2}

Python のバージョン 3.7 以上が必要。それ以前であれば

```
>>> r = subprocess.run(['echo', 'hello shell'], \
...                     stdout=subprocess.PIPE, stderr=subprocess.PIPE, encoding='utf-8')
...
などとする必要がある。
```

```
>>> r = subprocess.run(['echo', 'hello shell'], capture_output=True, text=True)
>>> print(r.stdout)
hello shell
```

基本的にはシェルコマンドを使わないでも Python だけで実現できることがほとんどであるが、シェルなら簡単にできるようなことであれば、このようにその実行結果を受け取って処理する方が話が早いこともあるだろう。例えば以下は先に示した関数 `count_py_lines1()` と全く同じ機能をシェルコマンドで実現する関数 `count_py_lines2()` の実装例である。

リスト 5 sample3.py (抜粋)

```
def count_py_lines2(dirname):
    print('*** count_by_lines2')
    # wc コマンドで行数を数える
    cmd = 'wc -l {}/*.py'.format(dirname)
    r = subprocess.run(cmd, shell=True, capture_output=True, text=True)
    # コマンドの出力からファイル名と行数を取り出す
    for line in r.stdout.split('\n'):
        l = line.strip().split()
        if len(l) == 2 and os.path.splitext(l[1])[1] == '.py':
            c = l[0]
            f = os.path.splitext(l[1])[1]
            print('{} : number of lines = {}'.format(f, c))
```

7.4 コマンドライン引数の処理

Unix のコマンドは多くの引数やオプションを受け取り、それに応じて異なる処理を実行する。Python では標準ライブラリの `argparse` を使うことで、コマンドラインで与えられた引数を簡単に処理することができる。使い方は簡単で、`argparse.ArgumentParser()` でオブジェクトを生成し、`add_argument()` でオプションを順次定義していけばよい。以下の例は `sample4.py` の抜粋である。

リスト 6 sample4.py (抜粋)

```
def parse_args():
    parser = argparse.ArgumentParser(description='Print Greetings')
    # 整数
    parser.add_argument('-i', '--integer', type=int,
                        help='number of output')
    # 文字列
    parser.add_argument('-g', '--greeting', type=str,
                        help='greeting')
    # 真偽値 (デフォルトは False)
    parser.add_argument('-c', '--capitalize', action='store_true', default=False,
                        help='capitalize or not')

    # デフォルトでは sys.argv をパース
    return parser.parse_args()
```

このサンプルでは `--integer` で指定された回数だけ `--greeting` で指定された文字列を出力する。また

--capitalize を指定されると、指定された文字列の各単語の先頭文字を大文字に変換する。以下はコマンドの具体的な実行例である。

```
$ python sample4.py --integer 3 -c -g 'hello python scripting'
*** command-line arguments
['sample4.py', '--integer', '3', '-c', '-g', 'hello python scripting']

*** parse results
option: integer => 3
option: greeting => hello python scripting
option: capitalize => True

*** show greetings
Hello Python Scripting
Hello Python Scripting
Hello Python Scripting
```

Unix の多くのコマンドでは長いオプションと、その省略形のオプションを使うことができるが、それはこの例でも同様である。例えば、11-12 行目は --integer とその省略形 -i を type=int で整数型として定義している。また 17-18 行目は --capitalize で指定される真偽値のオプションを定義しており、デフォルトでは False だが、オプションが指定されると True となるよう action='store_true' で指定されている。オプションの指定が終わった後にパーサーオブジェクトの parse_args() メソッドを呼ぶと自動的に sys.argv に格納されているコマンドラインオプションがパースされるので、後はこれを適宜用いればよい。parse_args() で返されるオブジェクトにはそれぞれのオプションがアトリビュートとして保存されているので、例えば --integer オプションであれば .integer アトリビュートを参照すればよい。

なお、この argparse を使うと自動的にヘルプメッセージが生成され、オプションとして -h や --help を指定すると以下のようにヘルプメッセージが表示される。

```
$ python sample4.py -h
usage: sample4.py [-h] [-i INTEGER] [-g GREETING] [-c]

Print Greetings

optional arguments:
  -h, --help            show this help message and exit
  -i INTEGER, --integer INTEGER
                        number of output
  -g GREETING, --greeting GREETING
                        greeting
  -c, --capitalize      capitalize or not
```

7.5 第 7 章 演習課題

参考:

- 課題 2 解答例
- 課題 3 解答例
- 課題 4 解答例

7.5.1 課題 1

サンプルを実行して動作を確認せよ。

7.5.2 課題 2

コマンドラインで与えられた (HTTP または HTTPS プロトコルの) URL にアクセスして得られた結果 (html) から title タグの内容を抽出して表示するプログラムを作成せよ。urllib.request.urlopen() を使って得られたバイト列を文字列に変換 (デコード) し、正規表現を用いて title タグの中身を抽出すればよい。サイトによって様々な文字コードが使われているので、chardet.detect() で推測したエンコーディングでデコードするのがよいだろう。例えば、以下のような実行結果が得られればよい。

```
$ python kadai2.py https://www.eps.s.u-tokyo.ac.jp/
東京大学 理学部 地球惑星物理学科・地球惑星環境学科／大学院理学系研究科 地球惑星科学専攻 |
$ python kadai2.py https://www.yahoo.co.jp/
Yahoo! JAPAN
```

7.5.3 課題 3

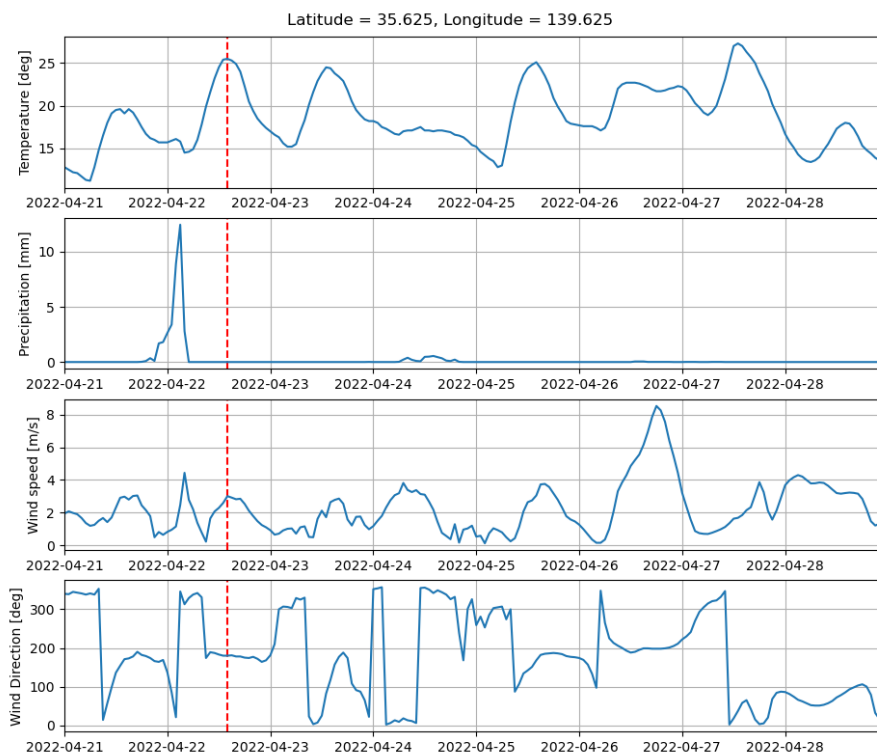
シェルの ls コマンドのようにカレントディレクトリに存在するファイルやディレクトリのリストを表示するプログラムを作成せよ。ただし、デフォルトではファイル名のアルファベット順 (ls のデフォルト) で、-S オプションが指定された場合はファイルサイズの降順で (ls -S と同じ順で) 表示するものとする。また、どちらの場合も -r が指定された場合には表示順を逆にせよ (ls と同じ仕様である)。例えば以下のような実行結果が得られればよい。

```
$ python kadai3.py -S
 2844 Fri Apr 22 13:42:47 2022 kadai4.py
 1707 Fri Apr 22 13:42:47 2022 kadai3.py
   763 Fri Apr 22 13:42:47 2022 kadai2.py
$ python kadai3.py -Sr
   763 Fri Apr 22 13:42:47 2022 kadai2.py
 1707 Fri Apr 22 13:42:47 2022 kadai3.py
 2844 Fri Apr 22 13:42:47 2022 kadai4.py
```

なお、この例ではファイル名だけでなく、ファイルサイズや更新日付も表示している。

7.5.4 課題 4

Open-Meteo というサイトは、無料で天気予報データをダウンロードできるサービスを提供している。決められた形式でサイトに HTTPS アクセスをすると、データが JSON 形式でダウンロードされる仕組みである。このサイトの東京の天気予報データにアクセスする URL を使おう。実行すると自動的にこの URL にアクセスし、ダウンロードしたデータ（プログラムを実行した時点での最新のデータ）を自動でプロットするプログラムを作成せよ。ただし、オプション無しで実行した場合には画面にプロットを表示し、`--save` オプションでファイル名を指定した場合には、指定されたファイル名（例えば `png` 形式など）でプロットをファイルに保存するようにせよ。例えば以下のようなプロットが得られればよい。この例の URL では温度、降水量、風速と風向きをプロットしている。（Open-Meteo のサイトではダウンロードするデータを自分でカスタマイズすることができる。）



ここで、`matplotlib` を使うのであれば、時間が `numpy.datetime64` 形式の配列になっていれば、横軸は自動でこの例のようにフォーマットされる。なお、このようなスクリプトで実行する際にはプロットした後に `matplotlib.pyplot.show()` を呼ぶと画面にウィンドウが表示される。画面に表示せずにファイルに保存したい場合は `matplotlib.pyplot.savefig()` を使えばよい。