Pythonの基礎: Major versionの違い

Ver 2.X	Ver 3.X
コンソール出力 print は組み込み構文のため、() がなくてもよい print “x=“, x	print が関数になったため、() が必要。 print(“x=“, x) 改行を抑制するには end = ‘’ 引数を与える print(“x=“, x, end = ‘’)
文字列で uniconde を指定する場合は u” を使う s = u’パイソン‘	文字列はuniconde に統一されているので、u” は不要
Ver3 .Xには Ver 2 =>Ver 3 変換ツールが含まれている 2to3 –w v2_script.py => バックアップファイル v2_script.py.bak をつくり、Ver 3のファイルが v2_script.pyに出力される 参考: https://www.python-izm.com/tips/2to3/	

PythonのTips

モジュールのインストール

参考: https://insilico-notebook.com/conda-pip-install/

Anacondaの場合、conda を使う方がよい (競合によるトラブルが少ない)

conda install module_name

一般の場合は pip を使う

pip install module_name

必要な環境変数設定

PATH=C:¥Anaconda3;C:¥Anaconda3¥Library¥mingw-w64¥bin;C:¥Anaconda3¥Library¥usr¥bin;C:¥Anaconda3¥Library¥bin;C:¥Anaconda3¥Scripts;C:¥Anaconda3¥bin;C:¥Anaconda3¥condabin

PYTHONPATH= 標準以外のモジュールのルートパス

Anaconda固有の環境変数

CONDA_DEFAULT_ENV=base

CONDA_EXE=C:¥Anaconda3¥Scripts¥conda.exe

CONDA_PREFIX=C:¥Anaconda3

CONDA_PROMPT_MODIFIER=(base)

CONDA_PYTHON_EXE=C:¥Anaconda3¥python.exe

CONDA_SHLVL=1

Pythonと他言語の比較

python	C	Perlなど
変数はすべてオブジェクト (実態は 辞書型 (連想配列) 変数)	**オブジェクト指向はC++などで拡張**	**オブジェクト指向は連想配列を使って疑似的に実現**
変数宣言 ・宣言構文はない。 ・値を代入したときに変数が生成され代入する値によって変数の型が決まる。 ・違う型の値の代入により、変数の型も変わる。 a = 10　整数型 a = 10.0　浮動小数点型 a = '10'　文字列型	・明示的な型宣言が必須。 ・変数の型は変更できない。 int a;　整数型 double a = 10.0;　浮動小数点型 char *a = '10' ;　文字列型	・明示的な宣言は必要ない。 ・my, local で明示的に宣言できる。 ・use strict; を使うことで、明示的な変数宣言を必須にできる。 ・代入する値によって変数の型が決まる。 ・値の代入により変数の型も変わる。 $a = 10 ;　整数型 my $a = 10.0;　浮動小数点型 my $a = '10' ;　文字列型
文区切り ・原則として改行で区切り ・文末に ; をつけてもエラーにはならない ・継続行を使うには、**行末に ¥ をつけ、改行コードをエスケープ**する ・括弧 (の後に改行をいれても、継続行と判断される。	・; を見つけると文の区切りになる	・; を見つけると文の区切りになる
ブロックの定義 ・インデントの大きさが同じ範囲が同一のブロックになる	・{} で囲まれた範囲が一つのブロック	・{} で囲まれた範囲が一つのブロック (pascalでは begin ~ end)

Pythonと他言語の比較

python	C	Perlなど

サブルーチン、関数定義
・def 文で関数を定義。
・同一インデントのブロック範囲が関数
・引数は関数定義で受け取る
・return文により戻り値を返すことができる
・関数自体は関数型の変数
def func_name(arg1, arg2):
sum = arg1 + arg2
return sum
	・関数型 関数名(引数) で関数を定義。
	・値を返さない関数は void型宣言をする
	・値を返す関数はreturn文が必須
	・引数は関数定義で受け取る
	・関数自体は関数型の変数
	int func_name(int arg1, int arg2) {
	int sum = arg1 + arg2;
	return sum;
	}
	void func_name(int arg1,) {
	printf("arg = %d¥n", arg1);
	}
		・sub 文で関数を定義
		・return文により戻り値を返すことができる
		・returnで戻り値を返さない場合、戻り値は undef になる
		・引数は関数内で @_ リスト変数として受け取る: 関数内でわかりやすい名前の変数に展開する方がいい
		・関数自体は関数型の変数
		sub func_name {
		my ($arg1, $arg2) = @_;
		$sum = $arg1 + $arg2;
		return $sum;
		}

PythonのTips: 局所変数と大域変数

参照: http://conf.msl.titech.ac.jp/Lecture/python/python-tips.html

変数の型とスコープは代入時に決まる。
・関数外で代入すると大域変数
・関数内で代入すると局所変数

【注意】関数内で、大域変数と同じ名前の変数に代入すると、その名前の局所変数が生成される。
関数内で大域変数へ代入する場合は、global 宣言を行う

global1.py

```
a = 1

def f():
    a = 2          局所変数を生成、大域変数は変えない
    print("in func:", a)

print("outer func1:", a)
f()
print("outer func2:", a)
```

実行結果

outer func1: 1	大域変数
in func: 2	局所変数
outer func2: 1	**大域変数**

global2.py

```
a = 1

def f():
    global a
    a = 2                    大域変数に代入
    print("in func:", a)

print("outer func1:", a)
f()
print("outer func2:", a)
```

実行結果

outer func1: 1	大域変数
in func: 2	大域変数
outer func2: 2	**大域変数**

基本的に、**大域変数は極力使わない方がいい**
多くの大域変数を使う場合、class宣言をしてオブジェクトの attribute として変数を宣言する
ただし、pythonのオブジェクトは辞書型を使うため、実行速度は遅い

Pythonのfor 文

一般的なプログラミング言語の for 文

BASIC: for i = 1 to 10 step 2

i を 1 から 10 まで、i に 2 を加えながら変えて、ブロックを繰り返し実行

C : for(int i = 1; i <= 10; i += 2)

i を 1 に初期化し、i に 2 を加えながら、i <= 10 の条件を満足している間、
ブロックを繰り返し実行 (perl, phpなど、pascal以降の言語の多くがこの形式を採用)。
BASIC, Fortranなどと同じ機能を持つが、より柔軟な繰り返し処理が可能。

```
int c = 0
for(int i = 1, k = 0; i <= 10 and k > 10; i += 2, k -= 1) {      # 複数のループ変数を使える
      c += i;
       if(k > -3) {
           i = 3;                                  # ループ変数をブロック内で変更できる
       }
 }
```

Python の for 文は、イテレータをリストとして渡す。

```
for v in list:
      a += v
```

BASICのような使い方をする場合、繰り返す値をすべて list 変数に入れて渡す。
数列のリスト変数を作製する際には、range() 関数を使うことが多い。

```
for i in range(0, 10, 2) :
      a += v[i]
```

for 文とリスト内包表記

Python の for 文:

```
a = []
for i in range(0, 10, 2) :
        a.append(v[i] + i)
```

これを 1行で書くことができる: リスト内包表記

a = [v[i] + i for i in range(0, 10, 2)]

- v[i] + i の計算を range(0, 10, 2) の各値 i について実行
- 実行した値を **リスト []** で返す

Python はインタプリタであるため、通常の文の実行は遅い
⇒ リスト内包表記の方が圧倒的に早い場合がある。

文字列の整形: .format と置換フィールド

参考: https://gammasoft.jp/blog/python-string-format/

Pythonの変数はすべてオブジェクト: 文字列型にもいろいろな attribute (メソッド) がある

a = 1.0
b = 1.0

・書式指定なし
print(“a=“, a, “ b=“, b)

・.format() と 置換フィールド {} で文字列を整形する
“a={} b={}“.format(a, b)　　=> “a=1.0 b=2.0” という文字列を返す
print(“a={} b={}“.format(a, b))　　=> print関数などに渡せる

・f”{変数名}” で文字列を整形する
print(f“a={a} b={b}“)

・f”{変数名}” で文字列を整形する
print(f“a={a} b={b}“)

・置換フィールド {インデックス番号:書式指定} で文字列を整形する
{2:.1f}　　2つ目の引数の値を、浮動小数点 (f) として小数点以下1桁に整形
{:12.4e}　次の引数の値を、指数形式 (e) で小数点以下4桁、最小幅12桁で整形

% を使った文字列の整形: C の printf形式

参考: https://note.nkmk.me/python-print-basic/

a = 1.0
b = 1.0

・C言語の printf() 関数と同じ書式も使える
注意: 変数の型によっては変換に失敗する。.format()を推奨

“a=%3.1f b=%3.2f“ % (a, b)　　=> “a=1.0 b=1.00” という文字列を返す
print(“a=%3.1f b=%3.2f“ % (a, b))　　=> print関数などに渡せる

%5d　　整数型で最低5桁
%12.4g　　浮動小数点型で小数点以下4桁、最低12桁
%s　　文字列型を右詰めで表示
%-s　　文字列型を左詰めで表示

Pythonのお約束: __name__ 変数

直接スクリプトが実行されたときのみ、main() 関数を実行する

Pythonでは、スクリプトファイルが実行されるとき、__name__ 変数が設定される。
- **スクリプトファイルがモジュールとして読み込まれる場合: モジュール名**
- **スクリプトファイルが直接実行される場合: '__main__'**

__name__変数の内容により、スクリプトが直接実行された場合とモジュールとして読み込まれた場合で、動作を変えることができる。

```
def main():
    print("executed directy")

if __name__ == '__main__':
    main()
```

とすると、モジュールとして読み込まれたときには main() は実行されないが
直接実行された場合は main() が実行される。

モジュールを開発している際に、モジュール単体を実行して動作確認をする場合などに
使う。

神谷作成 pythonプログラムの構造

実行方法: だいたい次のどれか

1. **開発初期や簡単なプログラム:**
   パラメータをプログラム中にハードコーディング。引数なしの実行のみ。
   python script.py
2. **頻繁にパラメータを変えるプログラム:**
   起動時引数で変えられる。引数なしの場合は初期値で実行する。
   実行時の最初 あるいは 最後に usage を表示するので引数なしで実行したのち、引数を変えて実行する。
   python script.py
   python script.py 1.0 2.0
3. **引数が複雑になった場合 I: 計算時間がかからないプログラム**
   実行時の最初 あるいは 最後に usage を表示するので引数なしで実行したのち、引数を変えて実行する。
   usage の表示に、実行例 ex: を表示するので、ex の文をコピペして実行してみるとわかりやすい。
4. **引数が複雑になった場合 I: 計算時間がかかるプログラム**
   引数なしで実行すると、usage と ex だけ表示して終了。
   ex の文をコピペして実行して動作確認したのち、引数の値を変えて実行

神谷作製 pythonプログラムの構造

ごく最近のプログラムの実行順は以下の通り

1. **モジュールの取り込み**
2. **プログラムの説明:** """ ~ """ コメント文で、プログラムの説明を記述
3. **一般的な大域変数の定義:** 物理定数など
4. **プログラム固有の大域変数の初期値の設定**
5. **matplotlib で描画するグラフのパラメータ**
6. **プログラム間で共通に使う小さい関数の定義:**
   pfloat, pint, getarg, getfloatarg, getintarg
   usage, terminate, updatevars
   terminate() は共通の終了処理をする: usage() を表示してから exit() など。
7. **起動時引数で初期値の更新**
   get(float/int)arg() を使うことで、引数が与えられていない場合は初期値を使う ということを1行で実行できる
   次の文をコメントアウトすると、引数がない場合には usage() を表示して実行を終了する。
   ```
   if len(argv) == 1:
       terminate()
   ```
8. **その他、プログラム固有の関数の定義**
9. **main() 関数の定義**
10. **直接スクリプトが実行されたときのみ、main() 関数を実行する**
    ```
    if __name__ == '__main__':
        main()
    ```

Pythonのエラー処理: try ~ except

Pythonのエラー処理はかなり厳しい

以下のように、頻繁に発生するエラーのすべてでプログラムの実行が終了する

- int(str) で文字列を整数型に変換しようとしたときに、文字列の書式が整数でなかった場合
- リストの要素を a[i] で取得しようとしたときに、i が a[] の要素範囲外だった時
- 未定義値の変数 (None) や変換できない型を含む演算

(str => floatなどの変換は自動的には実行されない)

参考: perl は、程度の低いエラーには寛容

- int(str) での文字列書式エラー: できる限り整数型に変換する。だめな時は 0 を返す
- リストの範囲外の要素にアクセスした場合: undef (未定義値) を返す
- undef や未定義の変数を含む演算: 未定義値は 0 として演算を実行
- 可能な限り、型の変換は自動的に行われる ("3.0" + 1 は 4.0になる)

Pythonでエラーが起こったときにプログラムを停止しないためには、try ~ except 構文を使う

まず try ブロックを実行し、エラー (例外、exception) が発生したら、exceptブロックを実行する

```
def pfloat(str):
try:
# float(str) を実行してみる
# str が浮動小数点として変換できる文字列であれば、変換した値を戻す
# str が浮動小数点として変換できない場合、例外を発生して except ブロックを実行する
        return float(str)
    except:
# str が浮動小数点として変換できない場合、None (未定義値) を返す
        return None
```

getarg() 関数

Pythonの問題: 起動時引数を取得する sys.arg リスト変数は、範囲外の index を渡すとエラーになってプログラムが終了する
Perlの場合 : リストに範囲外のindexを渡しても、undef が返ってくる。実行は継続。

引数で初期値を更新する関数 getarg() では、どのようなことが想定されるか

1. 何番目の引数か、**index (position) を渡し**、その値を return で返す
2. 引数が与えられない場合 ＝＞ 初期値をそのまま使う
   引数のindexの他、**初期値も渡す必要**がある
3. 引数が与えられた場合 ＝＞ 初期値を引数で置き換える
4. 初期値を与えない場合は None を返す
   初期値を与えないで getarg(position) を呼び出せるようにし、この際には初期値を Noneにする。
   “デフォルト引数” 機能を使う

```
# getarg()の引数としては、position と defval を渡す
# defval はデフォルト引数となっているので、
# getarg(position) と呼び出されたら、defvalに は Noneが代入される
def getarg(position, defval = None):
    try:
# sys.argv[position] が存在したら sys.argv[position] を返す
        return sys.argv[position]
    except:
# sys.argv[position] が存在しなかったら、defval を返す
        return defval
```

Pythonと他言語の比較: モジュール

Python	C	Perlなど
基本的な機能とモジュールの読み込み ・print() などは 組み込み関数 ・ほとんどの機能はモジュールを読み込むことによって使う **import numpy as np** numpyモジュールを読み込み、npという変数名でアクセスできるようにする **from math import log, exp** mathモジュールの中の関数のうち、log と exp 関数のみをインポートする **from matplotlib import pyplot as plt** matplotlibモジュールのpyplotモジュールをインポートし、pltという変数名でアクセスできるようにする	・低レベル関数のみ組み込み関数 ・printf()などの入出力関数なども組み込まれていない ・他はライブラィをリンクする。 ・ソースコードでは対応するヘッダーファイルを読み込み、変数、関数の型宣言を取り込む **#include <stdio>**	・print() などは 組み込み関数 ・ほとんどの機能はモジュールを読み込むことによって使う **use strict;** **require strict.pm;** ・モジュールを無効化することもできる **no strict;**
モジュール、ライブラリィファイル ・実行スクリプトとモジュールファイルには区別はない。一般に、どちらも拡張子 .py をつける ・モジュールに main() 関数がある場合、 if __name__ == '__main__': main() テクニックを使い、直接実行時のみ main() を実行するようにする	・ライブラリィのソースコードには main() 関数はあってはいけない ・ライブラリィファイルの拡張子も一般的に .c, .cpp などを使う ・ライブラリィはコンパイル後にリンクして実行可能ファイルを作製	・モジュールファイルの拡張子は一般的に .pm を使う

よく使うpythonモジュール

システム関係など

- sys
  sys.argvで起動時引数を取得
- csv
  CSVファイルの読み書き

科学計算関係

- math
  基本的な数学関数。sin, cos, tan, asin, log, exp　など
- numpy
  配列、行列を含む数値計算などにはほぼ標準。
  Python標準のリストより、numpy.ndarray を使う方がいい。
  numpy.ndarray からリストへは、ほとんど場合に自動的に型変換してくれる
  線形代数関数 (逆行列、一次連立方程式の解など)
- scipy
  numpyの機能に加え、信号処理 (FFT) など多様な数学関数がある

グラフ関係

- matplotlib
  グラフの描画

matplotlibでのグラフ表示

参考: http://conf.msl.titech.ac.jp/Lecture/python/matplotlib.html

```
fig = plt.figure(figsize = (8, 8))    # グラフサイズを指定して fig変数を取得
ax1 = fig.add_subplot(1, 1, 1)        # あとで複数のグラフ枠の描画もできるように、
                                        add_subplotを使い、最初のグラフ枠の変数を取得
ax1.plot(xarray, yarray)              # xarray, yarrayをデータの組とするグラフを描画する
```

一般的な場合

- matplotlibでは通常、グラフに表示するデータのリスト変数を plt.plot() で設定したのち、**plt.show()** でグラフを表示します。
- この場合、グラフウインドウを閉じるまで、プログラムが停止される。プログラムを終了するには、グラフウインドウを閉じる必要がある。

神谷の場合

- plt.pause(0.1) を使うと、グラフを最新のデータで表示したのち、プログラムの実行を継続する。0.1 は、pause() 関数を実行している際のsleep時間。なるべく短い時間に設定
- 放っておくと、プログラムが勝手に終了し、プログラムウインドウも閉じてしまうので、input() でプログラムが終了するのを止め、グラフを表示し続ける。

```
plt.pause(0.1)
print("Press ENTER to exit>>", end = '')
input()
```

- コンソールでENTERを押せばプログラムを終了できる
- plt.pause() を使うと、グラフをリアルタイムでupdateするプログラムも作れる