

免責事項

このワークショップは
アトリエのスタッフが作成したものであり
ソフトバンク公式のものではないことを
ご承知ください。



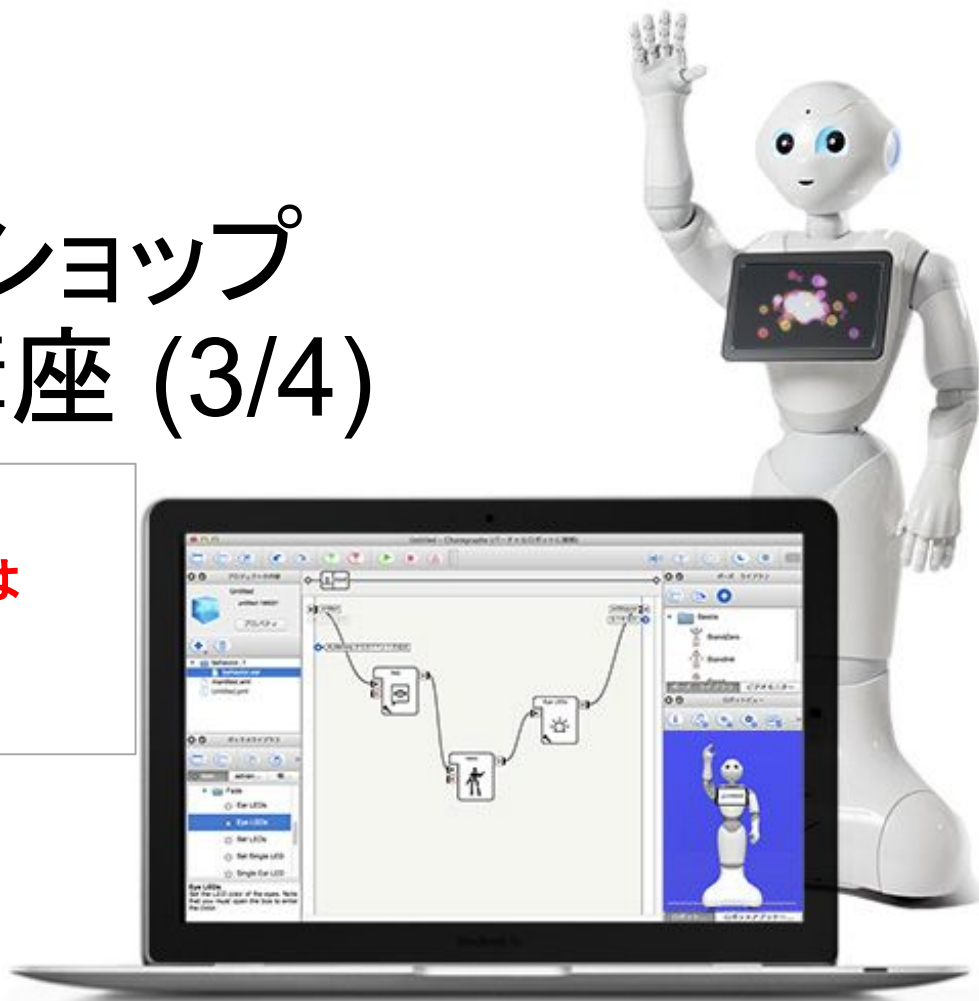
アトリエ秋葉原

Pepper ワークショップ Python 講座 (3/4)

Googleアカウントをお持ちでない方は
作成をお願いします！

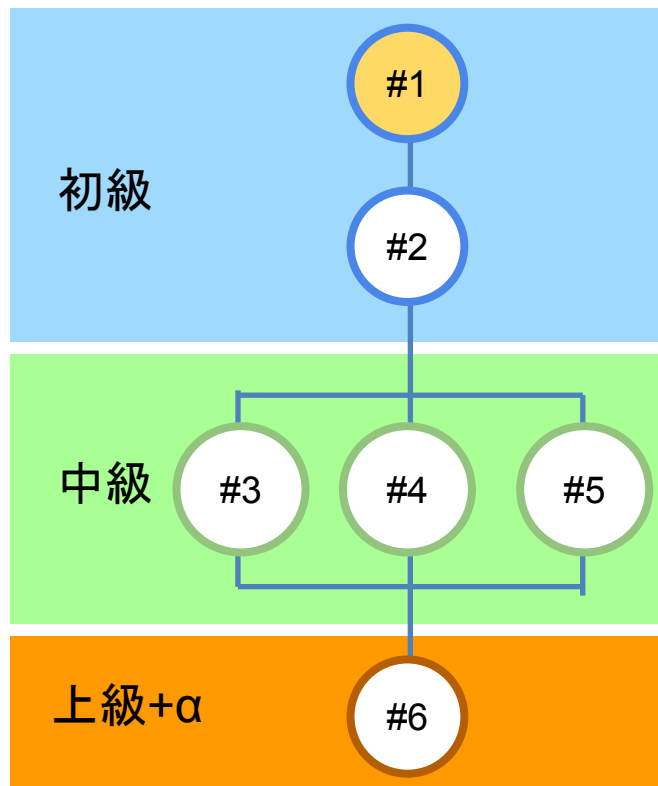
2018/1/07

Softbank Robotics



アトリエ秋葉原について

ワークショップ



タッチアンドトライ

自由に開発
質問はスタッフに
お客様同士の交流
検証や
打ち合わせの利用も可

1週間の予定

月	タッチアンドトライ
火	貸し切り(有料)
水	Pepper for Biz説明会 & タッチアンドトライ
木	貸し切り(有料)
金	タッチアンドトライ & ワークショップ
土日	タッチアンドトライ & ワークショップ

実体験とコミュニティで開発を促進する

アトリエ

コミュニティ



Pepperのアプリ開発という
実体験

相互
促進



経験や知見を
コミュニティで共有

実体験とコミュニティで開発を促進する



アトリエサテライト

有志でPepperと開発スペースを
提供している
企業、大学、コミュニティスペース

秋葉原で回答できない質問は
各サテライトへ

軽く自己紹介をしましょう！

- お名前
- 所属
- プログラミング経験や本日の意気込み

今回ワークショップ講師を務める
と申します。
よろしくお願いします

1.関数の作り方

2.オブジェクト指向について

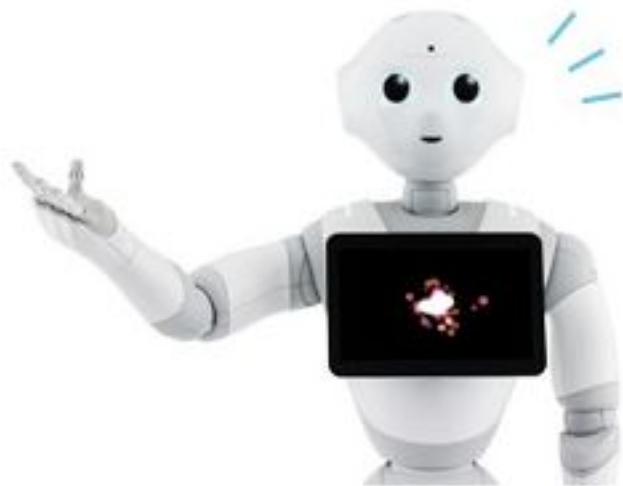
このワークショップでは
関数とクラス、それに付随して
変数の扱い方を詳しく学びます！

3.本WSの実行環境(colaboratory)

4.クラスについて

5.変数のスコープについて





関数を作ってみよう！

関数の書き方

～書き方～

```
def 関数名(引数):  
    処理を書く  
    (return 戻り値)
```

※関数がクラス内で定義する場合

1. 引数の先頭にselfをいれる
2. 関数を呼び出す時に
self.関数名とする

Choregrapheではあらゆるボックスがクラスで実装されているため、
Choregrapheで関数を定義する場合は必ず該当する

～例～

```
def onInput_onStart(self):  
    a = 10  
    b = 20  
    c = self.add(a,b)  
    self.logger.info(c)
```

```
def add(self, x, y):  
    sum = x + y  
    return sum
```

2つの値を足し算して出力する関数

演習問題1

6人のそれぞれの試験の点数が入っているリストがある。
リストを引数とし、最高得点と平均点を算出し、
以下の出力を得るような関数calcを作成しましょう。

出力結果: 最高得点は94点で、平均点は73点です。

```
def onInput_onStart(self):  
    score=[82,60,72,94,53,81]
```

```
def calc(???):
```

len()関数:
リストの要素数を返す組み込み関数

```
samp=[1,3,6,8]  
x=len(samp)  
#xに4が代入される
```

ヒント1

```
def onInput_onStart(self):  
    score=[82,60,72,94,53,81]
```

出力結果:最高得点は94点で、平均点は73点です。

```
def calc( ? ? ? ):  
    num=len(score) #リストの要素数から「人数」を取得  
    max=0    #配列の要素を順に見た時、暫定で最大の値を格納しておく変数  
    sum=0    #毎回のループにつき、ここに要素を足していく  
    for   
  
    ave=sum/num #総和を人数で割ることで、平均値を求める  
    self.logger.info(
```

解答例1

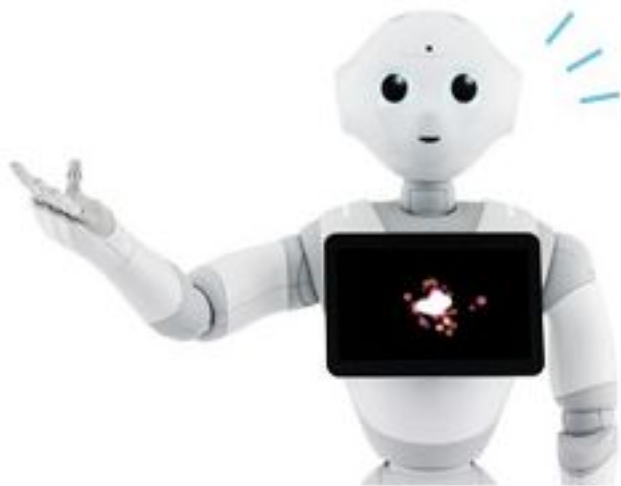
```
def onInput_onStart(self):
    score=[82,60,72,94,53,81]
    self.calc(score)

def calc(self,score):
    num=len(score)
    max=0
    sum=0
    for i in score:
        sum+=i
        if max<i:
            max=i
    ave=sum/num
    self.logger.info("最高得点は"+str(max)+"点で、平均点は"+str(ave)+"点です。")
    #self.logger.info("最高得点は{0}点で、平均点は{1}点です。".format(max,ave))
```

解答例2

```
def onInput_onStart(self):
    score=[82,60,72,94,53,81]
    self.calc(score)

def calc(self,score):
    num=len(score)
    max=0
    sum=0
    for i in range(len(score)):
        sum+=score[i]
        if max<score[i]:
            max=score[i]
    ave=sum/num
    self.logger.info("最高得点は"+str(max)+"点で、平均点は"+str(ave)+"点です。")
    #self.logger.info("最高得点は{0}点で、平均点は{1}点です。".format(max,ave))
```



**オブジェクト指向
について学ぼう！**

オブジェクト指向とは

オブジェクト指向はソフトウェア構成における一つの「**概念**」であり「設計思想」
(あくまで「考え方」であってこれ自体が仕組みを表すものではない)

カプセル化

- バラバラの機能を一つのまとまり(モジュール)として扱う
- モジュールの中のデータはブラックボックスにしておくという考え方

オブジェクト指向

ポリモーフィズム

- 複数の異なる型について共通のインターフェースを用意するのがよいという考え方
- 実装にはオーバーロードを用いる



実装の仕組みとしては「**クラス**」が一般的

オブジェクト指向を語る上で必須のキーワードたち

- オブジェクト・・・具体的な「モノ」のこと全般を表す
- クラス・・・「モノ」を生成するための設計図。型。
- インスタンス・・・クラスから生成された「モノ」
- メソッド・・・クラスに從属する関数
- フィールド・・・クラスに從属する変数
- コンストラクタ・・・インスタンスが生成されるときに最初に実行されるメソッド

普通はオブジェクト指向のプログラムを「クラス」を用いて実装するため、
「オブジェクトとインスタンスが同義」と認識しても問題ないことが多い

クラスとインスタンスについて

物理的なもののづくりに例えて説明すると・・・

クラス



ハンバーガー
レシピ



チーズバーガー
レシピ



Wチーズバーガー
レシピ



インスタンス



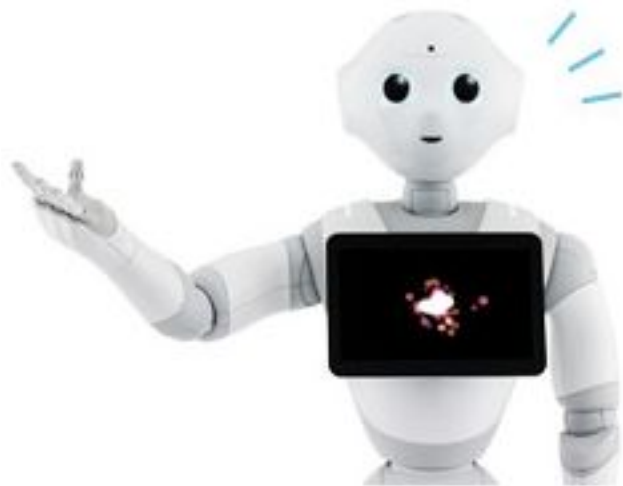
ハンバーガー



チーズバーガー



ダブルチーズバーガー



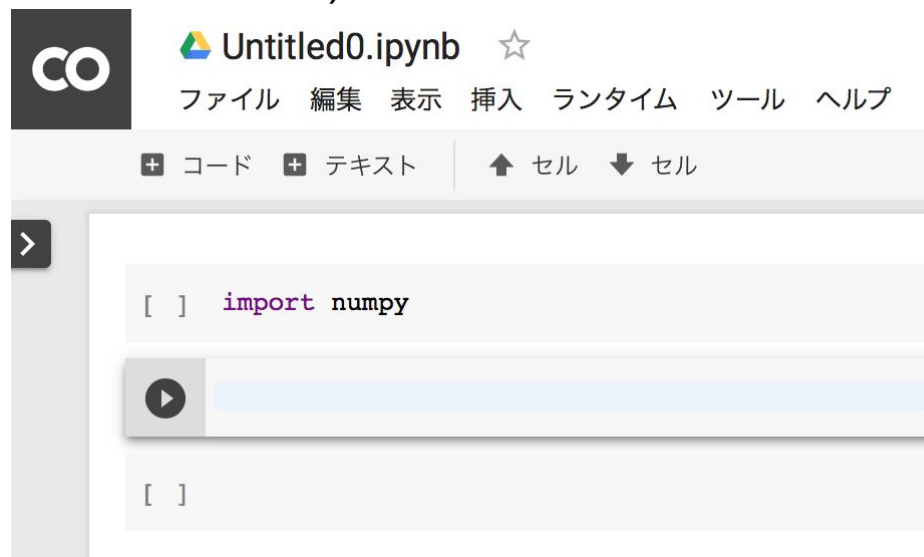
**Google Colaboratory
を使ってみよう！**

Google Colaboratoryとは

Googleが2018年から提供を開始したオンライン上のPython実行環境

メリットとして...

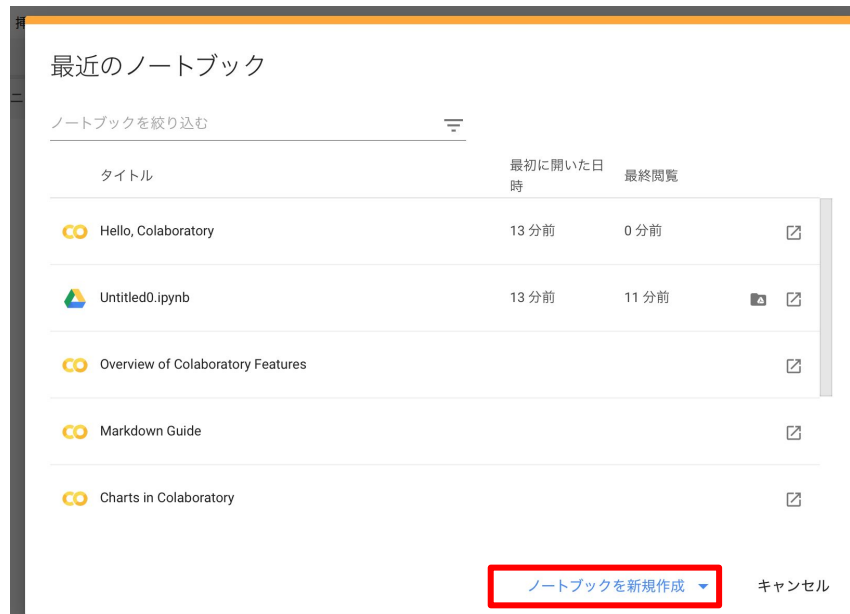
- 環境構築が不要！（自力でpythonインストールしなくて良い）
- googleアカウントですぐに使える
- 主要なライブラリはインストール済
- 機械学習に欠かせないGPUが使える
- 対話的実行が可能（1行ずつ実行できる）

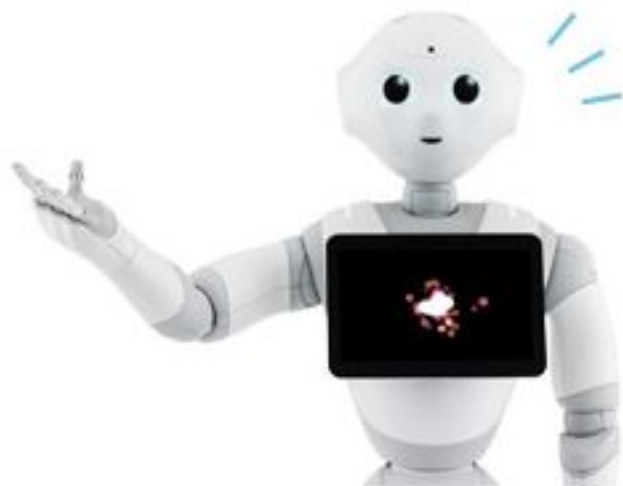


Google Colaboratoryの準備

- <https://colab.research.google.com/> にアクセス
("Colaboratory"で検索すると一番上に現れます)
- 「ノートブックを新規作成」
- 「Python2の新しいノートブック」

たったこれだけ。





クラスの作り方

Pythonでのクラス・インスタンスの実装

```
class クラス名:
```

```
    def 関数名(self, 引数1, 引数2,...):
```

```
        .....
```

```
インスタンス名 = クラス名()
```

```
インスタンス名.関数名(引数1,引数2,...)
```

クラスの宣言

メソッド(関数)の宣言

クラスに基づきインスタンス(モノ)
を生成し名前をつける

インスタンスの関数の呼び出し

具体例

```
class test_class:
```

```
    def sample(self, a):
```

```
        print("引数:" + a)
```

```
inst=test_class() #インスタンス生成
```

```
inst.sample("EXAMPLE")
```

文字列連結の"+"

クラス内の関数を使うにはまずインスタンスを生成
する必要がある(重要)

出力結果

引数:EXAMPLE

コンストラクタ・デストラクタ

```
class hamburger:
    def __init__(self):
        print("start loading")
        self.ing=[]
        self.ing.append("bread")
        self.ing.append("meat")
        self.ing.append("onion")
    def add_ing(self):
        self.ing.append("ketchup & mustard")
    def __del__(self):
        print("finish loading")
del show(self):
    print(self.ing)
del fetch(self):
    return self.ing

hb=hamburger()
hb.show()
hb.add_ing()
hb.show()
print(hb.fetch())
```

`__init__` 関数はクラスが**呼び出されたとき**に自動的に実行される関数。
これを**コンストラクタ**という。

ingの中身→[bread, meat, onion]

`__del__` 関数はクラスが**消滅するとき**に自動的に実行される関数。
これを**デストラクタ**という。

hb.ingの中身
→[bread, meat, onion, **ketchup&mustard**]



クラスの継承について

クラスの継承

クラスは料理のレシピに例えられる。

いまチーズバーガーのレシピを作りたい。ハンバーガーのレシピはすでに作成済み



チーズバーガーのレシピを1から書くよりも、
ハンバーガーのレシピにチーズを加えた方が早い



ハンバーガーレシピ
そのまま流用

+



これだけ追記

=



チーズバーガーレシピ

クラスの継承

```
class hamburger:
```

```
    def __init__(self):
```

```
        self.ing=[]
```

```
        self.ing.append("bread")
```

```
        self.ing.append("meat")
```

```
        self.ing.append("onion")
```

```
        self.ing.append("ketchup & mustard")
```



```
    def show(self):
```

```
        print(self.ing)
```

```
class cheeseburger(hamburger):
```

```
    def add_cheese(self):
```

```
        self.ing.append("cheese")
```



```
cbg=cheeseburger()
```

```
cbg.show()
```

```
cbg.add_cheese()
```

```
cbg.show()
```

class 子クラス名(親クラス名):

の形式でクラスを宣言することで、親クラスのメソッドが引き継がれた「子クラス」を作成することができる。これを**継承**という。

hamburgerクラスのメソッドを
継承したcheeseburgerクラスを宣言

ここでチーズを追加。
これでチーズバーガーのマニュアルが完成

チーズバーガーのマニュアルを元に
実際のハンバーガーを作成。
(インスタンス生成)

作ったハンバーガーにチーズを加えた

クラスの継承: オーバーライド

```
class hamburger:
```

```
    def __init__(self):  
        self.ing=[]  
        self.ing.append("bread")  
        self.ing.append("meat")  
        self.ing.append("onion")  
        self.ing.append("ketchup & mustard")
```



```
    def show(self):  
        print(self.ing)
```

```
class cheeseburger(hamburger):
```

```
    def __init__(self):  
        self.ing.append("cheese")
```

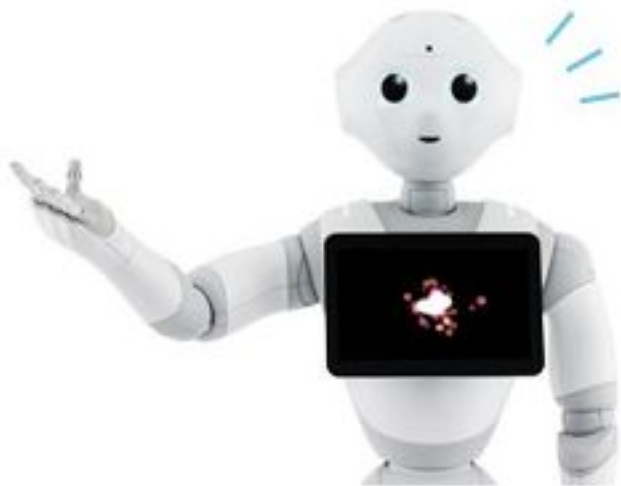


```
cbg=cheeseburger()  
cbg.show()  
cbg.add_cheese()  
cbg.show()
```

子クラスで定義された関数が親クラスと同じ名前を持つとき、
オーバーライドが発生。
親クラスで定義された関数は**無効化**され、
子クラスの関数に**置き換え**られる。

AttributeError: 'cheeseburger' object has no attribute 'ing'

`__init__` が上書きされたことで、`self.ing=[]` が無効になり、`append` できなくなったことがわかる。



変数のスコープ について

変数のスコープ ～関数編～

変数には有効な範囲があります。その範囲をスコープという。

実例その1: 関数の中で定義した変数

```
def scope():  
    a="one"  
    print(a)
```

```
scope()
```

```
print(a) #error
```

ここでエラーが出る理由:

実は変数aは**関数scope()の中でのみ**値を持っている！



関数scope()の外から変数aを呼び出しても**aの中身はからっぽ**

このように特定の場所でのみ値を持つ変数を **ローカル変数**という

```
def scopeb():  
    global b  
    b="one"  
    print(b)
```

```
scopeb()  
print(b)
```

bを**グローバル変数**として宣言。

これによって場所を問わずに変数bは値を持つようになる。

関数scope()の外から呼び出しても表示される。

変数のスコープ ～クラス編～

実例その2: クラス内の関数で宣言した変数

```
class scope2:  
    def cinsert(self):  
        c="hamburger"  
        print(c)  
    def cprint(self):  
        print(c)
```

ここで問題です。

```
sc2.cinsert()
```

```
sc2.cprint()
```

```
print(c)
```

はそれぞれ無事出力されるでしょうか？ エラーになるでしょうか？

```
sc2=scope2() #インスタンス生成
```

```
sc2.cinsert()
```

```
sc2.cprint()
```

```
print(c)
```

```
scope2.cinsert()
```

変数のスコープ ～クラス編～

実例その2: クラス内の関数で宣言した変数

```
class scope2:  
    def cinsert(self):  
        c="hamburger"  
        print(c)  
    def cprint(self):  
        print(c)
```

```
sc2=scope2() #インスタンス生成
```

```
sc2.cinsert()
```

```
sc2.cprint()
```

```
print(c)
```

```
scope2.cinsert()
```

ここで問題です。

```
sc2.cinsert()
```

```
sc2.cprint()
```

```
print(c)
```

はそれぞれ無事出力されるでしょうか？ エラーになるでしょうか？

無事出力

エラー

エラー

エラー

変数のスコープ ～クラス編～

実例その3: クラスの中で直接定義した変数

```
class scope3:  
    e="test"  
    def eprint(self):  
        print(e)
```

```
sc3=scope3() #インスタンス生成
```

```
sc3.eprint() #エラー
```

```
print(scope3.e)
```

```
print(sc3.e)
```

あるクラスの中で変数eを宣言して、
そのクラスの内側の関数で呼び出す。



実はうまくいかない。

では、同一クラス内で有効な変数を宣言するに
は??

self.変数名

変数のスコープ ～クラス編 "self.~"の意味～

実例その4: クラスの中でself.つきで定義した変数

```
class scope4:  
    def fprint(self):  
        self.f="test"  
        print(self.f)  
    def fprint2(self):  
        print(self.f)
```

scope4というクラスに從属する変数として f を宣言するという意味になる。

```
sc4=scope4() #インスタンス生成
```

```
sc4.fprint() #無事表示
```

```
sc4.fprint2() #無事表示
```

```
print(sc4.f) #無事表示
```

```
scope4.fprint() #エラー
```

注意！！

```
class scope4:  
    self.f="test" #これはだめ(宣言は関数内で行う)  
    def fprint(self):
```

同一クラス内の別の関数 から変数を呼び出すことができることを示している
(ただし、先にsc4.fprint()を実行する必要あり)

クラス内の関数を呼び出すにはまずインスタンスを生成する必要がある(おさらい)

変数のスコープ

実例その5: クラスの中でglobalつきで宣言した変数

前ページより

注意！！

```
class scope4:
```

```
    #これはだめ(宣言は関数内で行う)
```

```
    self.f="test"
```

```
    def fprint(self):
```

del()関数:

一度定義したグローバル変数を
白紙に戻す

```
class glo:
```

```
    global g
```

```
    g="test"
```

```
    def __init__(self):
```

```
        print(g)
```

} グローバル変数に関しては
(self.を必要としない場合)
「クラス内かつ関数の外」
に書いても一応動く。

```
glo=glo() #インスタンス生成
```

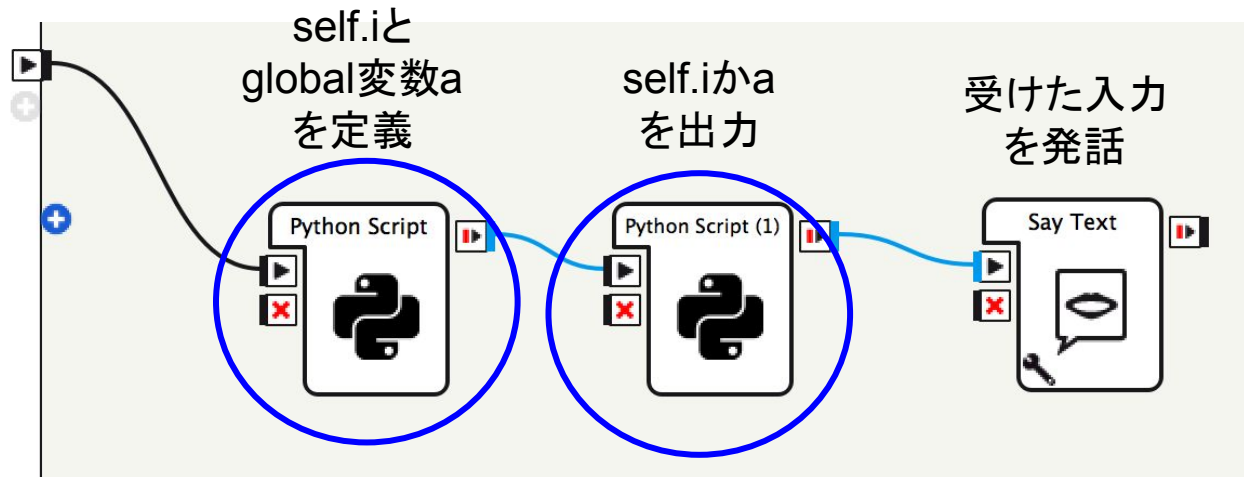
```
print(g) #無事表示
```

```
del(g) #gを白紙に
```

```
print(g) #エラー
```

Choregrapheへの応用

Choregrapheで配布ファイルの"variables.pkl"を開く。(バーチャルロボット使用可)



- self.で宣言した変数・・・スコープはボックスの中で完結
通常、線で値を他のボックスに受けわたす
- global変数・・・ワークスペース全体で変数の値を共有
(線で繋いでいなくても他のボックスから呼び出せる)


演習問題(おまけ)


Aさん～Eさんの5名についての出席表(辞書形式)を以下の要領で作ってみましょう。クラスで実装を行います。

- attendanceというクラスを作る
- その中にコンストラクタ、attend、showの3つの関数を作る
- コンストラクタは出席表(メソッド名は"table"とする)を生成。初期化する。
- 出席表初期値は{"A":False, "B":False, "C":False, "D":False, "E":False} とする
- attend関数
インスタンス名.attend("A,B,C")を実行したら、
そのインスタンスの出席表が
{"A":True, "B":True, "C":True, "D":False, "E":False}となるような関数
- show関数..."本日の出席表は、{ここに出席表表示}です"を出力。(print関数)

演習問題ヒント

```
class attendance:
```

```
    def __init__(  
        self.table= {"A":False, "B":False, "C":False, "D":False, "E":False}
```

```
    def attend(  
        attendlist=p.split(",")
```

入力("A,B,C")に対して、
["A","B","C"]とリスト化を行う。

```
    def show(  
        print(
```

self.tableの該当要素(出席者)の
値にTrueを格納

```
class attendance:
```

```
    def __init__( self ):
        self.table= {"A":False, "B":False, "C":False, "D":False, "E":False}

    def attend(self,p):
        attendlist=p.split(",")
        for i in attendlist:
            self.table["{}".format(i)]=True

    def show( self ):
        print("本日の出席者は{}です。".format(self.table))
```

Pepper デベロッパーポータル

「Pepper developer」で検索

<https://developer.softbankrobotics.com/jp-ja>

Pepperに関するデベロッパー向けの情報を集約したポータルサイト

- ・技術ドキュメント
- ・事例を共有するショーケース
- ・Pepper SDK for Android Studioのダウンロード
- ・最新ニュースの提供

Pepper アトリエ秋葉原 with SoftBank

「アトリエ秋葉原 ブログ」で検索

- ・ペッパー開発に役立つ記事を見ることができる
- ・イベントの紹介とイベントのレポートが見ることができる
- ・tipsの項目から開発に便利なツールを手に入れることができる

アトリエ秋葉原FBグループ

「アトリエ秋葉原 FB」で検索

- ・アトリエ秋葉原のFacebookグループです
- ・情報共有や質問ができます

Qiita

「Qiita pepper」で検索

- ・エンジニアの情報交換サイト
- ・PepperタグでPepperに関する様々な技術情報がある

おつかれさまでした！
これにてPython WS 3は終わりになります。
WSは続けてぜひ受講してみてください
お帰りの際はアンケートの記入に
ご協力ください

