python基本講座 1

python(標準入出力, コメント文, 変数)

print(“Hello World”)
→ Hello Worldが出力される（標準出力）

コメント文
“””
複数行にまたがったコメント
“””

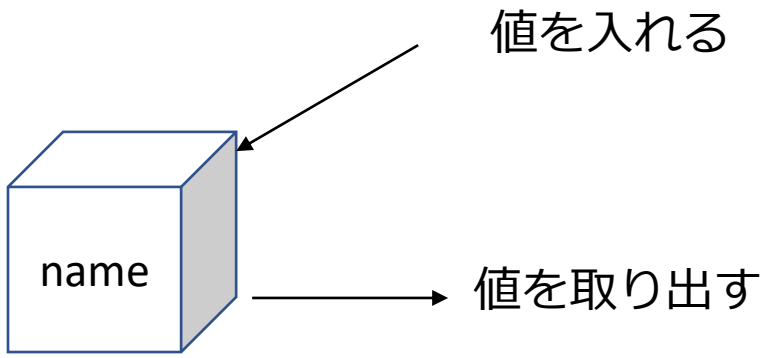
標準入力: input()関数

name = input(“あなたの名前は何ですか？”)
# nameという変数に標準入力の情報が入る

format関数: 標準出力の一部に変数に値を入れる
print(“My name is **{}**”.**format**(name))

b = ‘hello’ # bに文字列helloが入る
a = b = ‘hello’ # aとbに文字列helloが入る

変数って何なの？



“xxxx{}xxxx”.format(name) ← {}のところにnameが入る
(複数でも同じ)

python(変数)

変数に使える文字は、大文字・小文字のアルファベット、アンダースコアー(_)、数字、日本語のASCII文字。

animal = “dog” ←値を入れる
animal = input(“名前を入れてね”)　←標準入力

定数: 定数とは一度値を設定すると**変更してはいけない**変数。全て大文字で定義する
*) pythonの場合、厳密には定数というものを定義できない(あくまでお約束)
KING_OF_ANIMALS = “lion”

予約語: pythonのプログラム上あらかじめ決められている言葉（変数名などに使ってはいけない）
python3.7で33個ある(for とか tryとか)

python(論理型, OR, ANDについて)

論理型とは、**True**と**False**の二つの値を扱う変数の型です。

```
is_animal = True
# 変数is_animalにTrueを代入

is_animal = False
# 変数is_animalにFalseを代入

if is_animal:
    print(“動物！”)
# is_animalがTrueの場合のみifの中の処理を実行（pythonのif文はインデントで実行する処理を定義）
*)if文についてはのちの章で詳細に扱います

if is_animal OR is_bug:
   print(“生き物”)
# ORの場合、A OR B のA, Bどちらかがtrueの場合に処理を実行
if is_animal AND is_ghost:
  print(“動物霊”)
# ANDの場合、A AND BのA, Bどちらもtrueの場合に処理を実行
```

python(数値について)

pythonの数値型には、**int(整数型)**と**float(浮動小数点数型)**の２つだけで、数値を利用したい場合、計算をする場合に使用します。

```
age = 25
# 変数ageにint型の数値25が格納されます。
age += 1
# ageにage+1を入れます

 height = 175.5
# 変数heightにfloat型の浮動小数点数175.5が格納されます。

print(age + 1)
# age(25) + 1で26が表示されます。

print(height + 1)
# height(175.5) + 1で176.5が表示されます。

value = -5
# マイナスの数値
```

python(数値計算)

pythonの数値計算には以下の演算子を用います

+: 加算、 -: 引き算, *: 掛け算, /: 割り算(小数点以下計算), //: 割り算(小数点以下切り下げ), %(剰余), **(べき乗)

>>, <<: シフト演算

&, |: ビット演算(and, or)

12 & 21 = 01100 and 10101 = 00100 = 4

12 | 21 = 01100 or 10101 = 11101 = 29

python(数値について。2進数, 8進数, 16進数)

2進数は数値の前に**0b**、8進数は**0o**、16進数は**0x**をつけます。

```
age = 0b111
# 右辺は2進数のため、変数ageには7が格納されます。
age = 0o11
# 右辺は8進数のため、変数ageには9が格納されます。
age = 0x11
# 右辺は16進数のため、変数ageには17が格納されます。
```

#2進数の数値を文字列にする(文字列については次の動画参考)には、**bin**(数値)、8進数の場合は**oct**(数値)、16進数の場合は**hex**(数値)を用います。

```
print(bin(15))
# 2進数として'0b1111'が表示されます。
print(oct(15))
# 8進数として'0o17'が表示されます。
print(hex(15))
# 16進数として'0xf'が表示されます。
```

python(数値について。複素数)

#**複素数**は、**実数**と**虚数**を含んだ数値です。実数が3、虚数が2の場合、複素数として、**3 + 2j**と表示します(数Ⅲ)

```
a = 1 + 3j
# 実数1と虚数3の複素数
 b = 3 + 5j
# 実数3と虚数5の複素数

print(a + b)
# 複素数4+8jが表示されます。

print(a * b)
# 複素数-12+14jが表示されます。(複素数の掛け算)
```

python(数値について。複素数)

または、複素数を表す方法として**complex関数**を使う方法があります。例えば。実数が3、虚数が2の場合、複素数として、complex(3, 2)と表示します

```
a = complex(1, 3)
b = complex(3, 5)
print(a + b)
# 複素数4+8jが表示されます。
 print(a * b)
# 複素数-12+14jが表示されます。
```

#また、複素数型の実数と虚数をそれぞれ、取り出すには、real,imagを利用します。

```
print(a.real)
print(a.imag)
```

python(文字列型について)

文字列を変数として扱う時は、””,”で囲って変数に代入する。また、””””～””””とすると改行も自由に挿入して宣言することもできる。

```
fruit = “apple”
# 変数fruitに文字列appleが代入
 fruit * 10
# fruitを10回表示
fruit[2]
# fruitに格納した3番目の文字を表示

fruit = """apple
orange
grape
"""“
```

文字列メソッド

文字列の連携(+, *)、bytes型への変換(encode, decode)、文字列の回数検索(count)、文字列の始まりと終わりのチェック(startswith, endswith)、　文字列の除去(strip, rstrip, lstrip)、文字列の変換(upper, lower, swapcase, replace, capitalize)、文字列の検索(find, index, rfind, rindex)、文字列の条件 (islower, isupper)
文字列の一部(文字列[:])、format

メソッドの呼び出しについて注意点（メソッドはのちに詳しくやりますが、ここで簡単に説明します）

→のようにして呼び出す。　変数名(オブジェクト名)**.メソッド名()**

python(数値型⇔文字列型)

数値型から文字列型へ(str関数)

```
int_num = 10
float_num = 10.25
# 変数にそれぞれ、int型,float型を代入

print("int_num = " + int_num)
#文字列型 + int型では、エラーとなる
# str関数で解消
print("int_num = " + str(int_num))
print("float_num = " + str(float_num))
```

文字列型から数値型へ(int, float関数)

```
int_str = "10"
float_str = "10.25"
# 変数にそれぞれ、文字列型を代入

int(int_str)
# 文字列"10"が整数10に変換されます。
float(float_str)
# 文字列"10.25"が浮動小数点数10.25に変換されます。
```

python(リスト(配列)型)

リスト変数を使うと一つの箱の中に複数の値を入れることができる
（Javaなどの言語と違って、最初にリストのサイズを指定しなくてもよい）

```
list_a = [1, 2, 3, 4]
#  list_aに数値1,2,3,4を格納します。
 print(list_a[0])
#  0を指定するとlistの1番目の要素、1が表示されます。（indexは0から始まる）

list_a = [1, 2, "apple", 4]
#  list_aに数値1,2,4と文字列のappleを格納します。
 print(list_a[-2])
#  -2を指定するとlistの最後から2番目の要素、appleが表示されます。

list_a = [1, [0, 1, "apple"], 3, "lemon"]
#  list_aに数値1,3と文字列のapple,リスト[0, 1, "apple"]を格納します。

print(list_a[ 1 ][ 2 ])
# listの2番目の配列の3番目の要素、appleが表示されます。

list_a[ 1 ][ 2 ] = 'grape’
# listの2番目の配列の3番目の要素がgrapeになります。
```

python(リスト)

リストのスライス

list_a[0:2] # 1番目から2番目までの値
list_a[0:4:2] # 1番目から4番目まで1つ飛ばし
list_a[-3:] #最後から3番目以降

リストのメソッド

append: 値を一つ追加
extend: リストにリストを追加して拡張
insert: リストに位置を指定して値を追加
clear: リストを初期化
remove: 指定した要素をリストから削除
pop: 指定したインデックスの要素を取り出して削除
count:指定した値がリストに含まれる数を返す
index:指定した値のインデックスを返す
copy:リストをそのままコピーして新たなリストを作成し、返す

並び替え

sort: リストを並び替える（同じ型でないとエラー） sortedでもできる
reverse: リストの順番を入れ替える

python(辞書型（ディクショナリー））

pythonでは、以下のように宣言します。
dictionary = {‘キー1’: ‘値1’,’キー2’: ‘値2’}

car = { “brand”: “Toyota”, “model”: “Prius”, “year”: 2015 } # 辞書を作成します。

print(car[‘brand’]) #取り出せなかった場合エラー
print(car.get(‘brand’)) # 取り出せなかったNone
変数[‘キー’]または、変数.get(‘キー’)とすることで値を取り出すことができます。上の例では、Toyota

print(car.get(‘Model’, ‘Does not exist’))
# carにModelというキーが存在しない場合、 Does not existを返します

car.keys() # carのキー’brand’, ‘model’, ‘year’が返される
car.values() # carの値’Toyota’, ‘Prius’, 2015が返される
car.items() #carのキーとバリューがそれぞれが返される

for key,value in car.items(): # for文
    print("key = {}, value = {}".format(key,value)) # key,valueをそれぞれ取り出し、ループ内で回すことができます。
if key in car: # if文については後の章を参照
    print(car[‘key’])

python(辞書型（ディクショナリー）のメソッド)

```
car = { "brand": "Toyota", "model": "Prius", "year": 2015 }

car.update({'country': 'japan','prefecture': 'Aichi'})
# 辞書carにcountryとprefectureを値と共に追加します。

car['city'] = 'Toyota-shi'
value = car.popitem()
# 最後に追加した要素を削除します(この場合、city)、ただし、python3.7より前では、任意の値を削除します。
#またvalueに('city', 'Toyota-shi')とタプルにして返されます。

value = car.pop('prefecture')
# 指定したキーを削除します。また、上の場合valueにAichiが入ります。

car.clear()
# carに入った値が全て削除されます。

del car
# 辞書carが削除されます。
```

python(タプル)

タプルとは、**リストに似ていて**値を複数入れます。**値を変更、追加できない**

fruit = (‘apple’, ‘banana’, ‘orange’, ‘lemon’) # タプルの宣言

print(fruit[0]) # タプルの一番目の要素にアクセス

fruit[0] = ‘grape’ # TypeError, **タプルは値を変更することができない**

fruit = fruit + (‘grape’,) # タプルに要素を追加するには、 + (',)として、タプル+タプルの形にします。

# タプルのメソッドcount,index
# リストをタプルに変換する
list_a = ['banana','apple','grape’]
fruit = tuple(list_a) # ↑tuple(リスト)とすると、リストをタプルに変換することができる

リストと違う特徴は、値の変更ができないということ
{(‘A’, ‘B’): ‘value’}

- 配列よりもタプルの方が、アクセスするスピードが速い
- ハッシュ化して辞書型のキーとして、利用できる
- 値を変更したくないような値を用いる場合に、値が変更されないことを保障できる
（MONTH = (‘Jan’, ‘Feb’, ….)など）

python(セット)

セットは、リスト[]、タプル()と似たように複数の値を入れる入れ物です

- 同じ値を持つことがない（ユニーク）
- 順序が保持されていない（挿入された順番通りに取り出すことができない）
- ユニオンやインターセクションなどの集合処理を高速で行うことができる

set_a = {‘a’,‘b’,‘c’,‘d’,‘a‘} # setを作成、'a'が2つありますが、一つしか入っていません。

print(set_a) # set_aには、{'a', 'd', 'c', 'b'}の4つの要素があります。

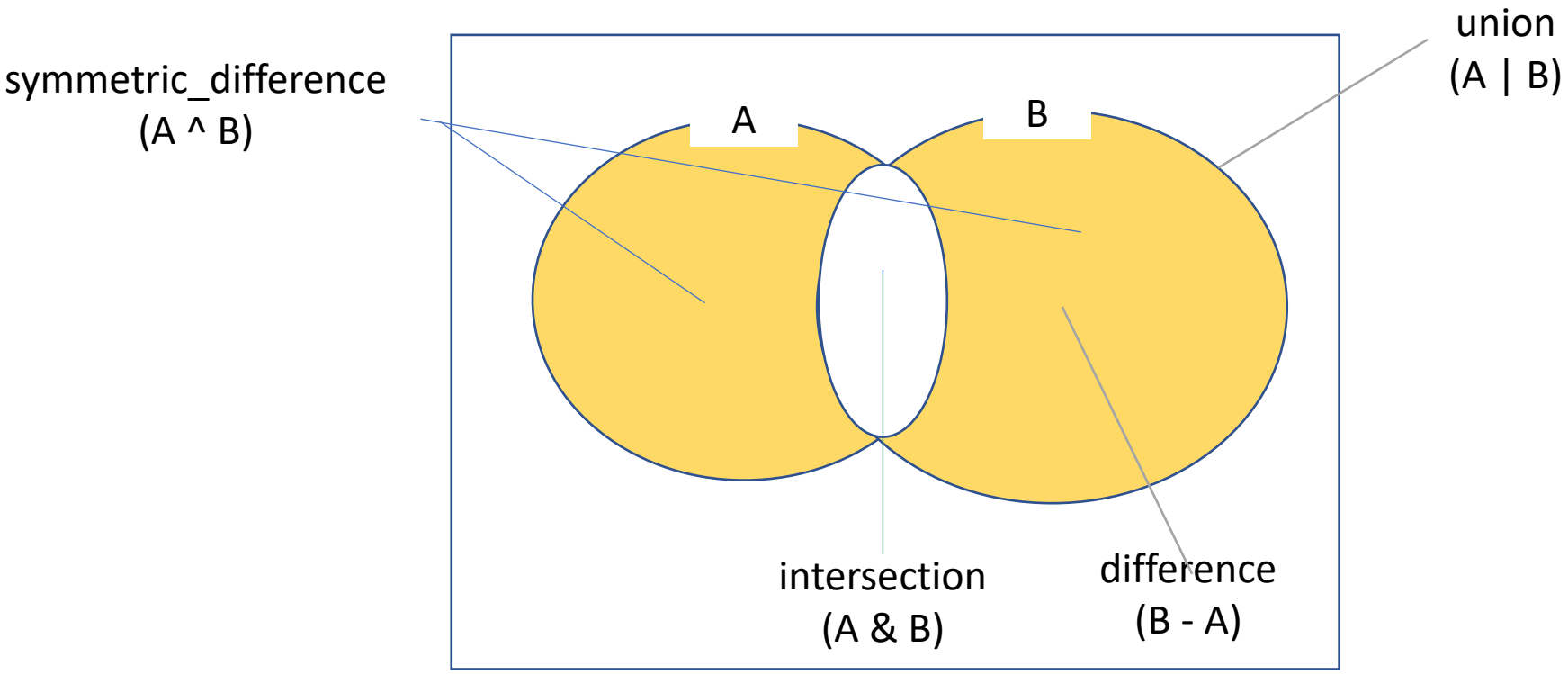
print('e' in set_a) # set_aには、'e'が入っていないため、Falseが返ります。
print('e' not in set_a) # set_aには、'e'が入っていないため、Trueが返ります。
print(len(set_a)) # set_aの数を返します。この場合4が返ります

set_a.add('e’) # set_aに'e'を追加します。
set_a.remove('e’) # set_aから'e'を削除します。'e'がない場合は、KeyErrorを返します。
set_a.discard(‘e’) # set_aから‘e’を削除します。‘e’がない場合でも、エラーは発生しません。
set_a.pop() # set_aから任意の要素を取り出して返して削除します。
set_a.clear() # set_aから全ての要素を削除します。

python(セットのメソッド)

セットのメソッドは、以下のようなものがあります

union(|)・・・ユニオン、和集合を返します。
intersection(&)・・・集合の共通する要素、積集合を返します。
difference(-)・・・片方の集合にあり、片方の集合にない要素、差集合を返します
symmetric_difference(^)・・・どちらか一方にだけある要素の集合を返します



python(セットのメソッド)

```
s = {'a', 'b', 'c', 'd'}
t = {'c', 'd', 'e', 'f'}

s | t # sとtのユニオンの集合、{'a','b','c','d','e','f'}を返します
s.union(t)  # sとtのユニオンの集合、{'a','b','c','d','e','f'}を返します

s & t  # sとtの積集合、{'c','d'}を返します
s.intersection(t) # sとtの積集合、{'c','d'}を返します

s - t #个s-t,s.difference(t)では、sとtの差集合、{'a','b'}を返します
s.difference(t)  #个s-t,s.difference(t)では、sとtの差集合、{'a','b'}を返します

s ^ t #s^t,s.symmetric_difference(t)では、sとtの片方に含まれる{'a','b','e','f'}を返します
s.symmetric_difference(t)  #s^t,s.symmetric_difference(t)では、sとtの片方に含まれる{'a','b','e','f'}を返します

s |= t #s = sとtのユニオン、つまりsにtの値を追加します。

s = {'apple', 'lemon'}
t = {'apple', 'banana', 'lemon', 'grape’}
u = {‘cherry’}
print(s.issubset(t))  # sの要素は総てtに含まれるため、sはtのサブセットでTrueを返します
print(t.issuperset(s))  # sの要素は総てtに含まれるため、tはsのスーパーセットでTrueを返します
print(t.isdisjoint(u))  # tの要素とuの要素は一つも被っていないため、Trueを返します。
```

python(変数、論理型、数値、文字列、リスト、タプル、辞書、セットの演習)

1. numという変数に数値の10を入れてください
2. numの型を標準出力してください
3. 2のnumを文字列に変換して、 num_strという変数に入れてください
4. リストnum_listにnum_strと'20', '30'を入れてください
5. num_listから新たな要素'40'を入れてください
6. num_listをタプルに変換してnum_tupleに格納してください
7. 標準入力を受け付けて、num_tupleに受け付けた標準入力を追加してください
8. セットnum_setを作成します。中身は'40','50','60'としてください
9. num_tupleとnum_setのユニオン（和集合）を表示してください（タプルをセット化するにはset(変数名)とします）
10. 辞書型、num_dictをキーnum_tuple、値num_strとして作成してください
11. リストnum_listの長さを表示してください
12. num_dictからキー'MyKey'を取り出して見つからない場合は'Does not exist'を返すようにして標準出力してください
13. リストnum_listに新たに'50','60'を一行で追加してください
14. 標準入力を受け付け、is_under_50という論理型変数に標準入力が50より小さいかどうかを入れてください
15. num_str = {num_strの値}として標準出力してください。
16. num_dictが持っているメソッドを標準出力してください。

基本講座１のまとめ

標準入出力、コメント文、変数、定数

論理型(True,False, AND, OR)、数値型(整数、浮動小数点数、2,8,16進数)、文字列型

リスト: []で宣言。

辞書: {キー: 値, キー:値}で宣言

タプル: ()で宣言。値の変更ができない

セット: {}で宣言。同じ値を入れることができない

python基本講座 2

python(制御文(if文))

if文は、渡された値の真偽を評価し、**真**の場合は実行され、**偽**の場合は実行されない式です

if **評価式**:
　式
とします。

pythonでは、以下の値が偽として認識されます

・None, False, 0, ‘’, 空のリスト,空の辞書,空のタプル,空のセット

[厳密には以下を満たすもの]
- 　__bool__()が定義されているクラスで、Falseが返されるもの

変数.__bool__() = bool(変数)

python(制御文(if文))

if 文を複数重ねるには、**elif, else**を利用します

等価性: <, <=, >, >=, !=, ==, not

```
if 評価式1:
    式 # 評価式1が正の場合実行
elif 評価式2:
    式 # 評価式1が偽で、評価式2が正の場合実行
else:
    式 # 評価式1も評価式2も偽の場合実行
```

・**and or not**

複数の条件式を評価する場合、and　orを利用します。また条件式の否定としてnotが利用されます

- 条件式1 and 条件式2とすると、条件式1と条件式2の**両方が正しい場合**に実行されます。
- 条件式1 or 条件式2とすると、条件式1と条件式2の**どちらか一つが正しい場合**実行されます。
- not 条件式とすると、条件式が**誤っている場合**に実行されます。

python(all, any関数)

if文でallとanyを利用すると処理の記載が楽になります。

 # allはオブジェクトの中が全てTrueの場合に処理をする
if all(反復可能オブジェクト): # **リスト、タプル等**、for in でループできる変数を入れます。
　処理

 # anyはオブジェクトの一部がTrueの場合に処理をする
if any(反復可能オブジェクト):

python(ループ文(for, while))

ループでは、複数回同じコードを実行します。pythonではfor, whileを用いてループを実現します

for 制御式:
　式 # ループ内の式が実行される

range関数(for i in range()として利用)
range(5): 0 1 2 3 4を返す
range(2, 6): 2 3　4 5 を返す
range(0, 10, 3): 0, 3, 6, 9

for i in range(5):
　print(i) # 0 1 2 3 4を表示

リストのループ
sample = [‘A’, ‘B’, ‘C’]
for s in sample:
　print(s) # A B Cを表示

python(ループ文(for, while))

enumerate関数: 配列の中の値とインデックスを同時に取得する

```
sample = [‘A’, ‘B’, ‘C’]
for index, value in enumerate(sample):
    print(index) # 0 1 2を表示
    print(value) # A B Cを表示
```

zip関数: 二つの配列の中の値を同時に取得する

```
for a, b in zip(list1, list2):
```

while: whileの中の式がTrueであるうちは処理を実行する

```
count = 0
while count < 10:
    print(count)
    count += 1
```

python(ループ文(continue, break, else))

pythonのループ内でのcontinue, break, elseの使い方

continue: ループ内にcontinueがあると処理が一度飛ばされます
break: breakが実行されるとループの外に処理が出ます

```
for i in range(10):
    if i == 3:
        continue # iが3の時は、処理が飛ばされます(i=4の場合の次の処理に移ります)。breakの場合は処理が終了してループの外に出ます
    print(i)
```

else: ループの外に出た際に実行されます(breakでループを抜けた場合には実行されません)

```
for i in range(10):
    print(i)
else:
    print('ループ終了') #ループ終了後実行されます
```

python(セイウチ演算子(3.8以降))

:= イコールの前に:コロンとつけた演算子

変数の代入と変数の使用を同時に実行できるという特徴を持っている

```
if (n := 10) > 5: # nに10を代入するのとn > 5の比較を同時に実行している
    print('nは5より大きい')
```

一番よく使うパターンは、while文だろう（実務でセイウチ演算子を利用した経験はないので推測）

```
n = 0
while (n := n + 1) < 10:
    print(n)

n = 1
while n < 10:
    print(n)
    n += 1
```

python(例外処理(try, except))

例外処理: 実行時に発生するエラー（**実行時エラー**）を制御して処理を行う

```
try:
    処理
except エラー名:
    例外発生時の処理
```

例)

```
try:
    a = 10 / 0
except ZeroDivisionError:
    print('例外処理')
```

FileNotFoundError: プログラムで指定されたファイルが見つからないエラー
IndexError: 配列などで指定したインデックスに値が存在しないエラー
TypeError: 型に関するエラー
ZeroDivisoinError: 0で割ろうとしたことによるエラー
Exception: すべての例外

python(例外処理(try, except, else, finally))

例外処理を複数つないだもの

```
fruits = ['apple', 'orange', 'grape']
count = 12
try:
    fruit = fruits[2]
    print(fruit + count)
except IndexError as e:
    print('配列に存在しないインデックスを指定した場合')
except TypeError:
    print('文字列型+数値型等、型の誤ったものと計算した場合などに実行')
except Exception:
    print('全ての例外をキャッチ')
else:
  処理
finally:
  処理
```

else, finally: 例外処理後に実行される

else: 例外が発生しなかった場合に実行され、例外が発生した場合には実行されない
finally: 例外が発生した場合にも、発生しなかった場合にも実行される

python(例外を返す(raise))

```
def devide(a, b):
    if(b == 0):
        raise ZeroDivisionError('bを0に設定しないでください')
        #raiseをすると呼び出し元にメッセージと共に、例外を返すことができます。
    else:
        return a / b

try:
    result = devide(10,0)
    #関数を呼び出した結果、ZeroDivisionErrorが返ってきます。
except ZeroDivisionError as e:
    print('{} : {}'.format(type(e), e))
    #ZeroDivisionErrorが発生したため、<class 'ZeroDivisionError'> : bを0に設定しないでくださいと表示

try:
    code
except Exception as e:
    raise ○○ from e

class MyException(Exception): # 例外自作
```

python(関数)

関数とは一連の処理を1つにまとめたもの。pythonの関数は以下のように定義する

def 関数名():
  処理

```
def print_hello(): # 関数の定義
    print("Hello") # 関数の処理

print_hello() # 関数の呼び出し、Helloが表示されます。
```

関数に引数を与えるには
def 関数名(arg1, arg2):

```
def num_max(a, b):
    if a > b:
        return a
    else:
        return b

num = num_max(10, 20)  # 大きい方の20が返ってきます
num = num_max(b=20, a=10) # 引数を直接指定することもできます
```

python(関数)

関数の**デフォルト値の**指定は以下のようにします
def function(arg1, arg2=100):

可変長引数: 場合に応じてその都度、引数の長さが変わるもの
*arg1のように*を一つつけると、**可変長のタプル**
arg2のように*を２つつけると、可変長の辞書**

```
def spam(arg1, *arg2):
    print("arg1 = {}, arg2 = {}".format(arg1, arg2))  # arg1とarg2を表示します。
    print(type(arg2)) # <class 'tuple'>が表示され、arg2がタプルであることがわかります。
spam(3,4,5,5)  # 1つ目の引数はarg1,残りはarg2に4,5,5がタプルとして挿入され、"arg1 = 3, arg2 = (4, 5, 5)"が表示されます
```

```
def spam(arg1, **arg2):
    print("arg1 = {}, arg2 = {}".format(arg1, arg2)) # arg1とarg2を表示します。
    print(type(arg2)) # <class 'dict'>が表示され、arg2が辞書型であることがわかります。
spam(3,arg3=5, arg4=6) #"arg1 = 3, arg2 = {'arg3': 5, 'arg4': 6}"が表示されます。
```

複数の返り値
return a, b

python(グローバル変数)

変数には利用できる領域**(名前空間)**があります。別の名前空間の変数を更新することができない場合があります。
例えば、以下のように関数の中から関数の外の変数を書き換えることができません。

globalを使わない場合##
```
def printAnimal():
    animal = 'cat'
    # ここで宣言した変数animalは関数の外で宣言したanimal(‘dog’)とは異なる変数として 扱われ、この関数内のみで有効な変数です 。(animal = 'cat'としても関数の外のanimalは書き換わりません)
    print(animal) # print(animal)でcatが表示されます。

animal = ‘dog’ # 関数外の変数
printAnimal()
print(animal)  # このprintでは、関数の外で有効なanimal('dog')が表示されます。 上から実行した結果、 cat,dogと表示されます。
```

globalを使った場合、変数の名前空間を広げることができる#####################################
```
def printAnimal():
    global animal #animalの前に、globalを付けて、global変数として宣言するとこのanimalは 関数の外までスコープが広がります(同名の変数が関数の外に存在する場合は、同じ参照先になります)
    animal = ‘cat’#このanimalはグローバル変数のため、関数の外のanimalの値も変更されます。
    print(animal)# print(animal)でcatが表示されます。

animal = 'dog' # 関数外の変数
printAnimal()
print(animal) # 関数実行時にanimalの値はcatに書き換えられているため、ここでcatと表示されます。
```

python(inner関数とノンローカル変数)

pythonでは、**関数の内部に関数**を書くことができます。これを**inner関数**と言います。また、**ノンローカル変数(nonlocal)**として宣言すると内側の関数から外側の関数で宣言された変数を書き換えられるようになる

・Inner関数の書き方

```
def outer():
  def inner():
    # inner関数の処理
  # 外の関数から、inner関数を呼び出す等、行う。
```

inner関数の用途

- 外の関数の外からアクセスできないような関数を作成する
- 関数の中で同じ処理が何度も発生する場合に、一つにまとめる
- デコレータ関数（後述）を作成する

・nonlocal変数を用いたinner関数の定義

```
def outer():
    outer_value = '外' # 外側の変数outer_valueに外という文字が代入されます。

    def inner():
        nonlocal outer_value  # nonlocalと宣言することで、外側の関数の変数であるouter_valueを変更 できるようになります。
        inner_value = '内'  # 内側の関数の変数のinner_valueに内という文字が代入されます。
        outer_value = '内側で変更' # outer_valueはnonlocalとして宣言しており、このouter_valueを用いて 外側の関数の変数outer_valueに文字列'内側で変更'が代入されます。

    inner() # inner関数を実行します。
    return outer_value  # return outer_value(この場合、文字列'内側で変更'が返されます)

value = outer() # 関数outerを呼び出して、その返り値('内側で変更')をvalueに代入します。
print(value) # value('内側で変更')を表示します。
```

python(ジェネレータ関数)

ジェネレータ関数は以下のように**yield**を用いる

```
def func():
  yield 〇〇
```

- ジェネレータ関数を宣言して実行すると、yieldの部分で処理がストップし、yieldの〇〇に記載された値（配列、辞書等オブジェクト）が呼び出し元に返されます。
- その後、再度ジェネレータ関数を呼び出すと、yieldの部分から処理がスタートし、yieldの部分でストップします。
- プログラムが終了するまで、何度もジェネレータ関数を呼び出すことができます。（その度に、yieldからスタート、yieldで終了します）

```
def generator(max): #ジェネレータ関数の宣言
    print("generater created")
    n = 0
    while n < max:
        yield n # 呼び出し元にnを返し処理がストップ
        print("yield called")
        n += 1

gen = generator(10) # ジェネレータの作成、この時点では処理は走りません。
a = next(gen) # generatorを呼び出します。generator createdと表示されて、yieldの部分でとまり、yeild nのn（この場合0）が値として返ってきます
print(a) # 0が表示されます。
```

python(ジェネレータ関数)

ジェネレータ関数には以下のメソッドがあります

send(): yieldで停止している箇所に値を送ります。
throw(): 指定した例外が発生して処理が終了させます。
close(): ジェネレータを正常終了させます。

gen.send(10) # yieldの部分に、sendを創出し、yieldの部分の値が10となって処理が進む。

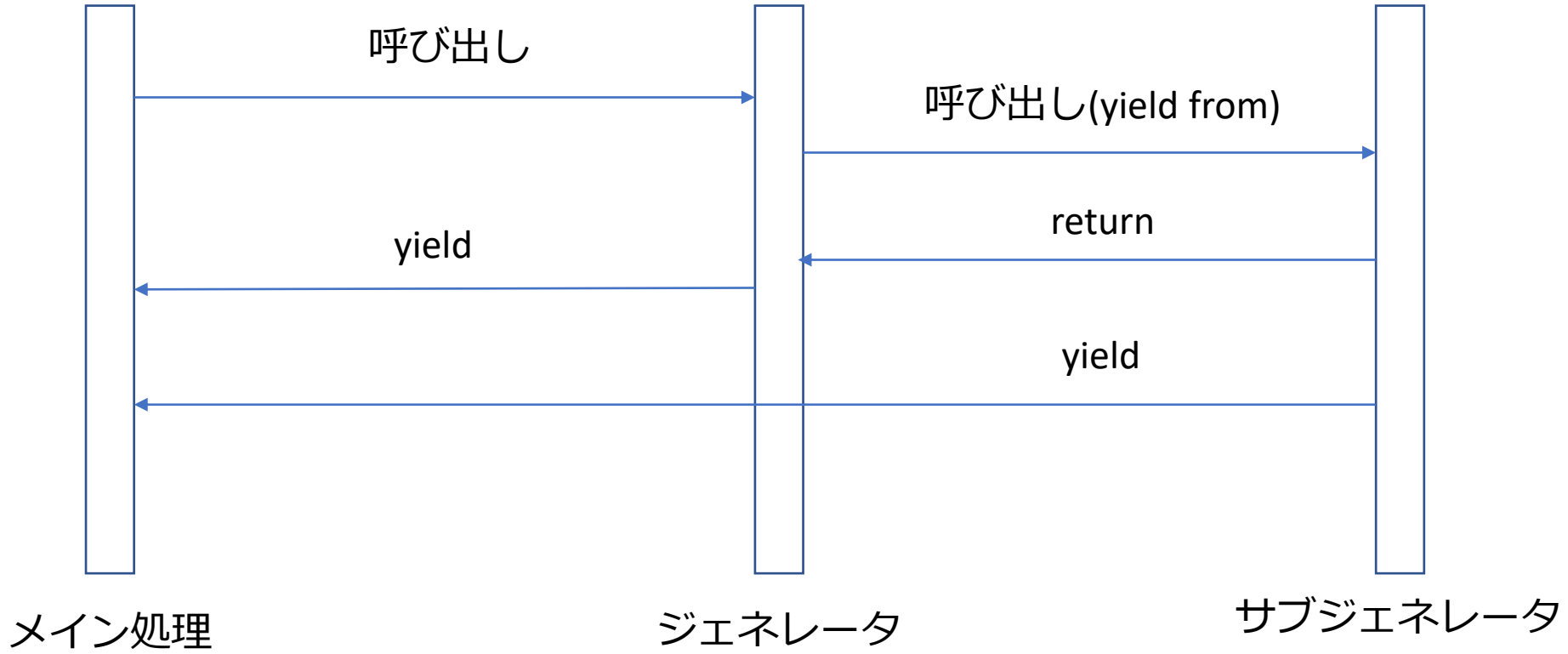
```
for v in gen:
    print(v)
    if v > 2:
        gen.throw(ValueError("Invalid Value")) # ValueErrorを送出して処理を終了します。
```

gen.close() # close()をするとジェネレータ処理が強制終了して、以後、generatorはStopIterationと表示されて利用できなくなります。

python(サブジェネレータ)

pythonでは、ジェネレータからジェネレータを呼び出すことができます(このとき、呼び出し元がジェネレータ、呼び出し先がサブジェネレータ)
このとき、**yield from**と記述して、ジェネレータからサブジェネレータを呼び出します。

- 呼び出し先のサブジェネレータからさらに別のジェネレータを呼び出すこともできます
- サブジェネレータにreturnを追加することで、呼び出し元(ジェネレータ)に値を返すことができます。
- サブジェネレータでyieldを実行すると一番最初の呼び出し元に値を返します。



python(ジェネレータ何に使うの)

これまでジェネレータについて勉強しましたが、結局これって何に使うの？と思われる人もいるかと思います。

ジェネレータの一番の使い道は**メモリ使用量を削減**することです。

例えば以下の場合、実行される結果は同じですが利用される**メモリ使用量が違います**。

```
list_a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in list_a:
    print(i)


def num(max): # generatorを定義
    i = 0
    while i < max
        yield i
        i += 1

for x in num(10): # generatorを呼び出し
    print(x) 0 から9まで表示
```

一番、generatorの出番だと思えるのはDBから大量のデータを引っ張るときでしょう。
DBの大量のレコードを一気に取得して何万行の配列にすると大変メモリを食います。
この場合、generatorを使うと使用するメモリをほぼ0にすることができます
（ただし、generatorの方が処理は遅いので、使うかどうかは場合にもよります）

python(高階関数)

pythonでは、関数もオブジェクトの一つにすぎないため**変数として扱うこともできます**。また、関数を他の関数の引数として渡したり、返り値として扱うこともできます。
関数を引数にしたり、返り値にする関数を**高階関数**と言います

- 関数を変数のように扱う

```
def print_hello():
  print("hello")

var = print_hello  # 変数varに関数をオブジェクトとして代入します。
var() # このように変数に()を付けると関数が呼び出され、helloが表示されます。
var2 = ['apple', 'orange', print_hello]  # 配列の3番目（インデックスだと2）にprint_helloをオブジェクトとして、入れておきます。
var2[ 2 ]() # 配列のインデックス2のprint_helloを呼び出すと、helloと表示されます。
```

- 引数、返り値に関数を利用するやり方

```
def print_world(msg):
    print("{} world".format(msg))
def print_konnichiwa():
    print("こんにちは")
def print_hello(func):
    func("hello") # 引数funcを関数のように利用します。
    return print_konnichiwa # 関数print_konnichiwaを返します。

var = print_world # 変数varに関数print_worldをオブジェクトとして代入します。
a = print_hello(var)
# 関数print_helloの引数をprint_worldにして、呼び出します。
# func("hello")がprint_world("hello")になり、関数で"hello world"が表示されます。
# また、aには、関数print_konnichiwaが格納されます。
a() # 関数print_konnichiwa()が実行され、"こんにちは"と表示されます。
```

python(lambda式)

# ifを1行で表して代入する
x = 0 if y > 10 else 1

pythonで1行で終わるような関数を用いる場合には、**lambda式**という無名関数を用いることがよくあります。lambda式は以下のように書きます。

lambda 引数: 返り値

lambda_a = lambda x: x * x # aに無名関数のオブジェクトを代入します。
↓
```
def func(x):
    return x * x
```

res = **lambda_a**(12) # 引数を12として、無名関数を実行します。戻り値は、x * xの144が返ってきます。
print(res) # 144が表示されます。

lambda_b = lambda x, y, z=5: x * y *z # lambda式で、複数の引数を取り、一部の引数にデフォルト値を入れて定義します。
print(**lambda_b**(3,4)) # x=3,y=4として関数が実行され、60が表示されます。

- 条件式を用いたlambda

c = lambda x, y: y if x < y else x # x<yの場合yを返し、それ以外はxを返す

python(再帰)

再帰とは、関数から自分の関数を呼び出すことです。

```
def sample(a):
    if a < 0:
        return
    else:
        print(a)
        sample(a-1)
```

フィボナッチ数列
1, 2のとき: 1
3以上のとき: 直前の二つの数値の和
1,1,2,3,5,8,13,…

とする数列

```
def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

python(リスト内包表記)

ループと条件を使って、1行でリストを作成する方法（リスト内包表記）

変数名 = [式 for 変数 in リスト (if 条件式)]

```
new_list = []
for x in list_a:
    new_list.append(x)
```

↓

```
new_list = [x for x in list_a]
```

```
new_list = []
for x in list_a:
    if type(x) == int:
        new_list.append(x)
```

↓

```
new_list = [x * x for x in list_a if type(x) == int]
```

ジェネレータを返すには、以下のように宣言します

x = (リスト内包表記)

```
value = [x for x in range(10000)] #リスト
tup = tuple(x for x in range(10000))#タプル
gen = (x for x in range(10000)) #ジェネレータ
set = {x for x in range(10000)} # セット
```

python(デコレータ関数)

Pythonの場合、@関数名と関数の前につけることで、デコレータ関数を利用します。
デコレータは、関数オブジェクトを引数にとって引数にとった関数に実行時に変更を加えます
デコレータは関数間で、ある処理を共通に利用したい場合によく用いられます

```
def my_decorator(func): # デコレータ関数
    def wrapper(*args, **kwargs):
        print('*' * 100)
        func(*args, **kwargs)
        print('*' * 100)
        return 1
    return wrapper

@my_decorator # デコレータ関数の利用
def func_a(*args, **kwargs):
    pass

@my_decorator # デコレータ関数の利用
def func_b(*args, **kwargs):
    pass

func_a↓
def wrpper(*args, **kwargs):
    print('*' * 100)
    func_a(*args, **kwargs)
    print('*' * 100)
    return 1
```

基本講座２(まとめ)

制御文・・・if, elif, else, or , and , not

ループ文・・・for, while, range関数, enemurate関数, zip関数, continue, break, else, セイウチ演算子

例外処理・・・try, except, else, finally

関数・・・引数、返り値、可変長タプル、可変長辞書、グローバル変数（global）、inner関数、ローカル変数

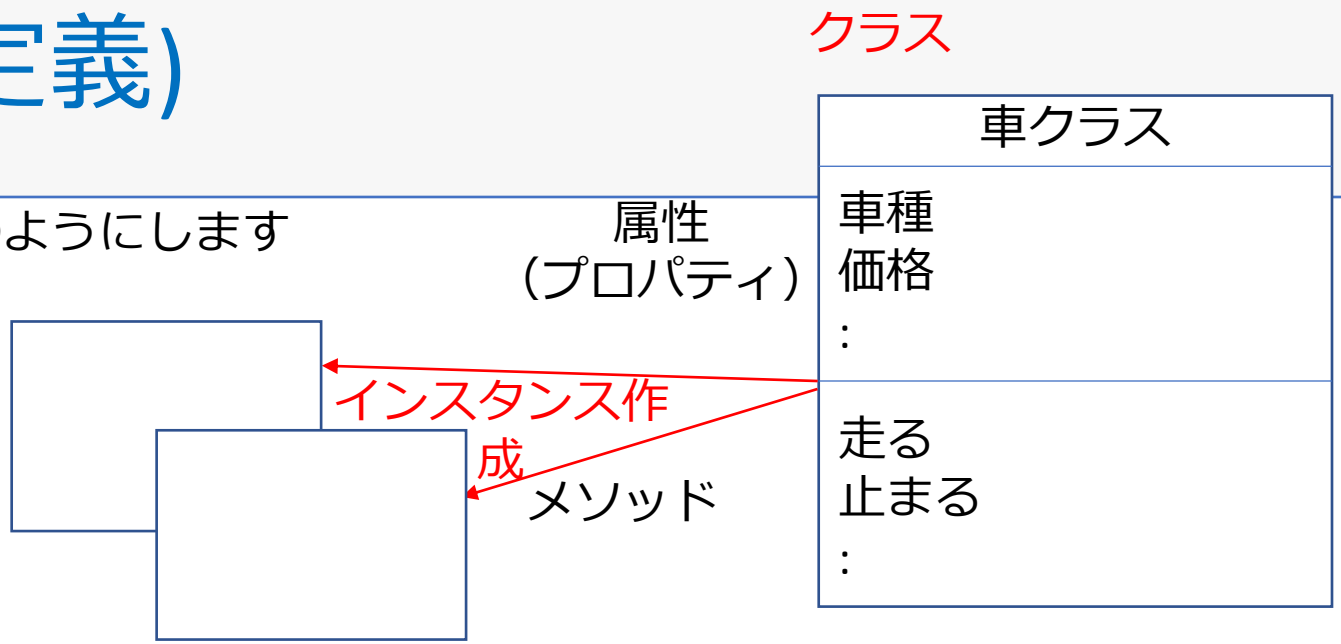
ジェネレータ関数・・・yield, サブジェネレータ, yield from

高階関数、lambda式、再帰、デコレータ関数、リスト内包表記

次はオブジェクト指向プログラミングをやっていきます

python基本講座３
（オブジェクト指向）

python(クラスの定義)



pythonでクラスを定義するには、以下のようにします

```
class クラス名:
 """ドキュメンテーション文字列"""
 def メソッド名(self, ...):

class Prius:
    maker = "Toyota"
    tire = "Studless"
    wiper = "Standard"  # クラスと、そのプロパティの「maker」「tire」「wiper」を定義しています。
    def my_func(self): # クラスのメソッド

myCar = Prius()  # インスタンス変数名 = car()とすることで、インスタンスが作成されます（Prius()でもOK）
print(myCar.maker)  # インスタンス名.プロパティ名とすることで、プロパティの値が表示されます。#(上の例だとToyota)

mySecondCar = Prius() # 新たにインスタンスを作成することもできます。上のmyCarとは別物です。

classes = ["apple", "grape", Prius]
myFamilyCar = classes[2]()
print(myFamilyCar.maker) # toyotaと表示
myFamilyCar.my_func() # myFamilyCarインスタンスのmy_func関数を実行する
```

python(クラス変数、インスタンス変数)

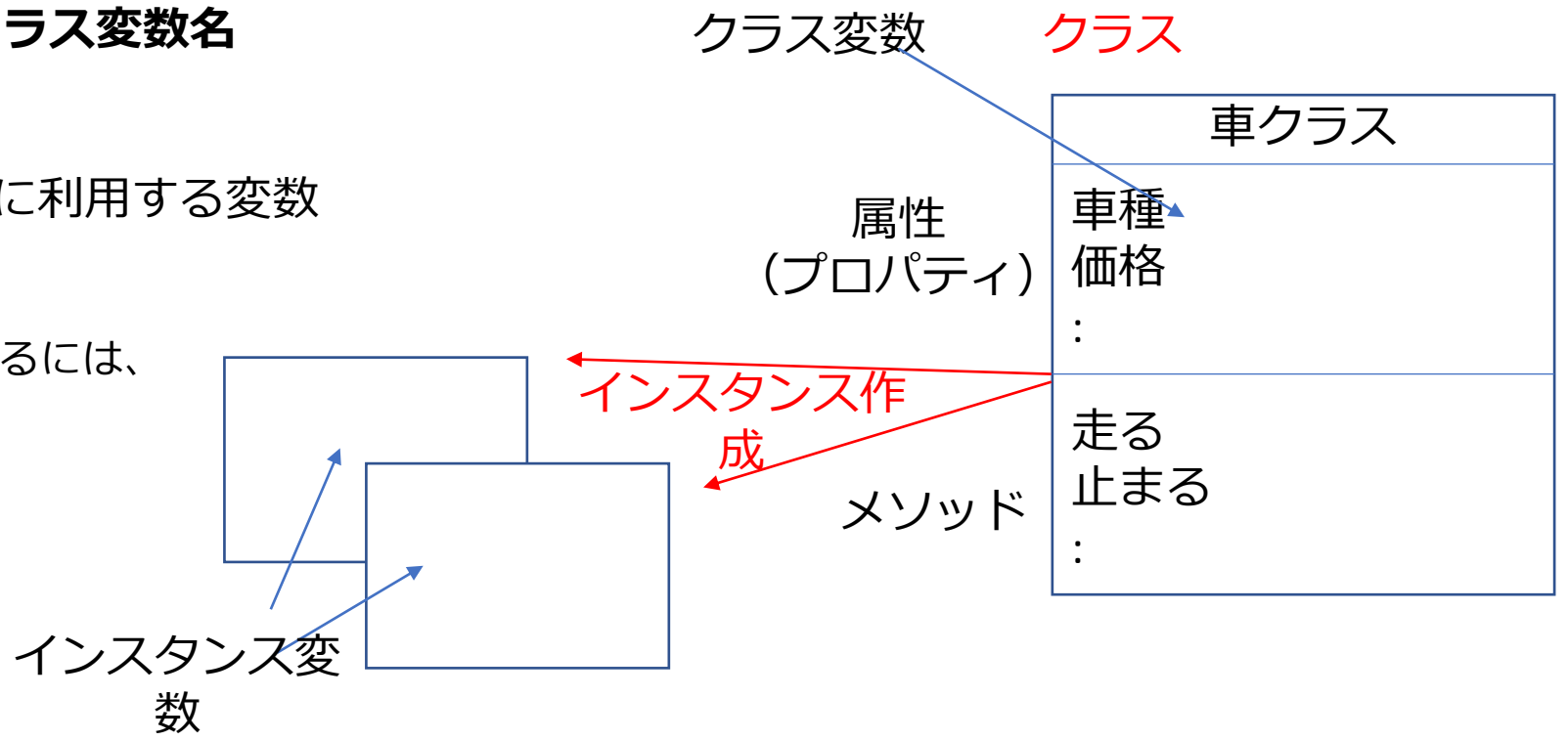
pythonには、クラスで定義する属性（プロパティ）として、クラス変数とインスタンス変数があります。

<クラス変数>
- オブジェクト同士で共有することができる変数
- メソッドの内部でなく、クラスの直下に記載

クラス変数にアクセスするには
クラス名.クラス変数名
インスタンス名.__class__.クラス変数名

<インスタンス変数>
- インスタンスごとに別々に利用する変数
- メソッドの内部に記載

インスタンス変数にアクセスするには、
インスタンス名.変数



python(コンストラクタ(__init__))

オブジェクトをインスタンス化する際に呼び出されるメソッドをコンストラクタと言います。pythonの場合、__init__というメソッドを作成すれば、インスタンス作成時に、自動的に呼び出されます。

例えば、クラスのプロパティを初期化する際などに、コンストラクタを利用すると便利です。

```
class sample_class:
    def __init__(self, msg):
        print("コンストラクタが呼び出されました")
        self.msg = msg # プロパティmsgをコンストラクタの引数msgで初期化します。

    def print_msg(self):
        print(self.msg)  # プロパティmsgを表示します。

sample = sample_class("Hello World“, )
# インスタンス作成時にコンストラクタ__init__が実行されます。
#"コンストラクタが呼び出されました"と表示されたのち、プロパティmsgに"Hello World"が格納されます。

sample.print_msg()
# インスタンスsampleのprint_msgメソッドを実行します。プロパティmsgは"Hello World"なので、print(self.msg)で"Hello World"と表示されます。
```

python(デストラクタ(__del__))

デストラクタ(__del__)を定義すると、インスタンスを削除する際に呼び出されます。

```
class del_sample:
    def __del__(self):
        print("delが呼び出されました。")

del1 = del_sample()
del del1
#del インスタン名とすると、インスタンスが削除されます。
#インスタンスを削除すると、__del__が呼び出され、"delが呼び出されました。"が表示されます。
```

python(インスタンスメソッド、クラスメソッド、スタティックメソッド)

Pythonのクラスのメソッドには、インスタンスメソッド、クラスメソッド、スタティックメソッドの３種類があります。

インスタンスメソッドとは
クラスから作成したオブジェクト(インスタンス)を用いて呼び出すメソッド

↓インスタンスメソッドの例

```
class MyClass:
    def __init__(self, name):
        self.name = name
    def print_hello(self): # インスタンスメソッド
        print('Hello ' + self.name)

AClass = MyClass('World')
AClass.print_hello()
```

引数のselfは必ずつけます。selfは自分自身(作成したインスタンス自身)を表し、例の場合だとコンストラクタで初期化した、インスタンス変数nameを呼び出しています(self.name)

python(インスタンスメソッド、クラスメソッド、スタティックメソッド)

クラスメソッドとは
クラスをインスタンス化せずに実行できるメソッド

↓クラスメソッドの例(@classmetthodをつけます)

```
class MyClass:
    class_name = 'myclass'
    def __init__(self, name):
        self.name = name
    @classmethod
    def print_hello(cls):
        print('Hello ' + cls.class_name)

MyClass.print_hello()
```

・@classmethodで定義
・第一引数をclsを取ります(clsはクラス自身)

cls.変数名とすることで、クラス変数に**アクセスできます**。初期化（インスタンス化）がされていないため、インスタンス変数に**アクセスすることはできません**

python(インスタンスメソッド、クラスメソッド、スタティックメソッド)

スタティックメソッドとは
インスタンスメソッドやクラスメソッドのようにインスタンスやクラスが引数に渡されることはしません。

↓スタティックメソッドの例(@staticmetthodをつけます)

```
class MyClass:
    class_name = 'myclass'
    def __init__(self, name):
        self.name = name
    @staticmethod
    def print_hello():
        print('Hello World’)

MyClass.print_hello()
```

- **@staticmethodで定義**
- **クラス変数へもインスタンス変数へもアクセスできない**

ちなみに、インスタンスからクラスメソッド、スタティックメソッドを利用することはできる

python(特殊メソッド)

インスタンスにアクセスする際に、特定の処理を実行すると呼び出されるメソッド。例えばa + bとして実行するとき、内部的にはa.__add__(b)として呼び出される。

__str__: str(object), print(object)の際に呼び出され、オブジェクトを文字列として返す

__bool__: if文で論理値を返すことができる

__len__: len()実行時に呼び出される。

__eq__: == の際に呼び出される

__hash__: 文字列をハッシュ値として返す。この関数を定義することで、クラスを辞書として扱うときやsetに要素を入れるさいに利用する

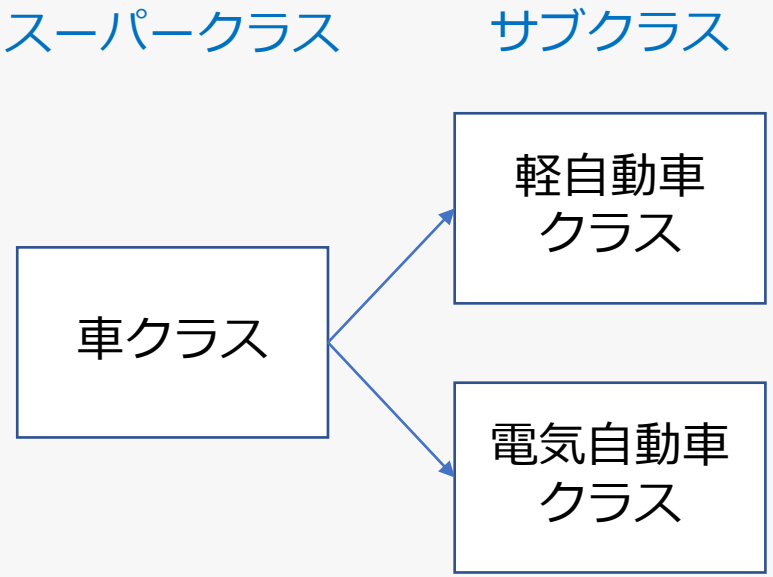
__name__: クラスの名前を表す

その他の特殊メソッドについては、以下の資料を参照
https://docs.python.org/ja/3/reference/datamodel.html

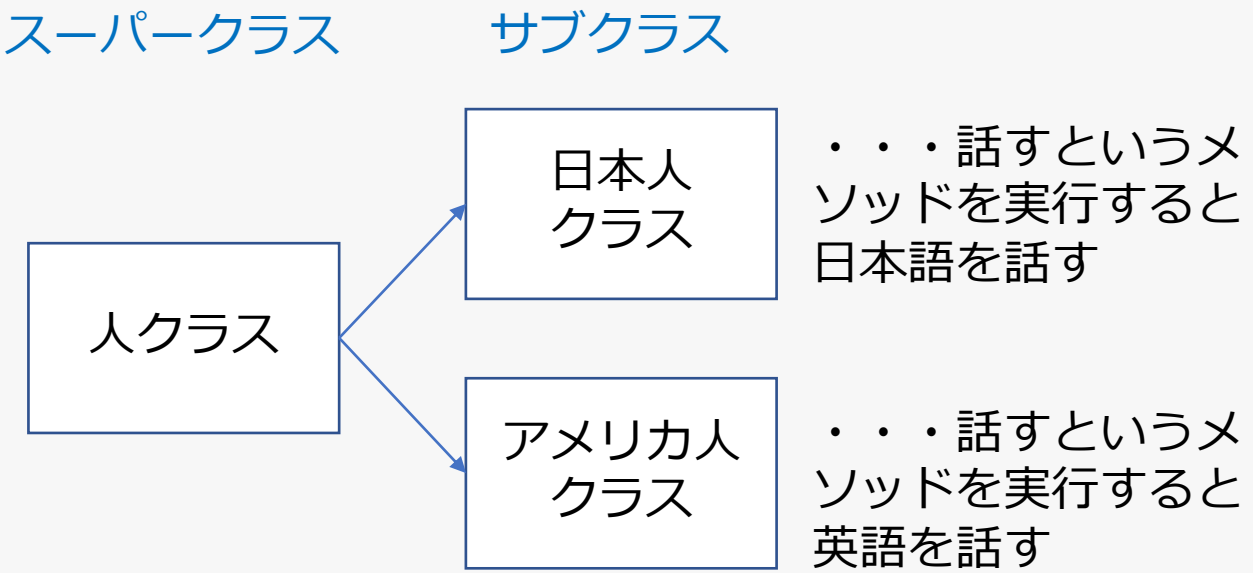
オブジェクト指向

継承: ある別のクラスからそのクラスの持っている性質を引き継ぐこと。車というクラスから軽自動車というクラスを新たに作る場合を継承という。この場合、軽自動車というクラスには車クラスと同じ**プロパティ**、**メソッド**を引き継ぐことができる。この場合継承元のクラスをスーパークラス、継承先にクラスをサブクラスと言う。

ポリモーフィズム: サブクラスを複数作成して、サブクラスごとに同じ名前のメソッドをそれぞれ作成して、処理の中身を変える。呼び出す際には、中身を意識せずに実行できる性質のこと



継承



ポリモーフィズム

python(クラスの継承)

クラスを継承する場合には、
class クラス名(継承先): と記述します

```
class Person:
#「人」クラス人としての機能（プロパティ、メソッド）を持つ
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def greeting(self):
        print("Hello! I'm {}".format(self.name))

    def breathing(self):
        print("su-- ha--")

class Employee(Person):
# クラスPersonを継承すると、親クラスのプロパティ(name,age)、メソッド(greeting,breathing)をEmployeeクラスのもののように自由に扱えるようになります。
    def __init__(self, name, age, number):
        super().__init__(name, age) # super()とすることで、親クラスを呼び出し、__init__でnameとageを初期化できます。
        self.number = number

    def greeting(self):
        print(“Hello Sir My name is {}. number is {}”.format(self.name, self.number)) # greetingを**オーバーライドして、**新たなメソッドとして定義しています。
# ちなみに、**オーバーロード**は引数や返り値が異なるが名称が同一のメソッドを複数定義すること(Pythonでオーバロードはできません)
    def greeting(self, msg):
    def serve(self):
        pass  # Employeeクラスでserveメソッドを新たに定義します。ここでは、詳細は省略します。
```

python(クラスの多重継承)

クラスを多重継承する場合には、
class クラス名(継承先1, 継承先2): と記述します

```
class ClassA:
    def __init__(self, name):
        self.a_name = name
    def print_a(self):
        print("a = {}".format(self.a_name))
    def print_hi(self):
        print("A hi")

class ClassB:
    def __init__(self, name):
        self.b_name = name
    def print_b(self):
        print("b = {}".format(self.b_name))
    def print_hi(self):
        print("B hi")

class NewClass(ClassA, ClassB):
    def __init__(self, a_name, b_name, name):
        ClassA.__init__(self, a_name)   # 親クラスClassAの初期化
        ClassB.__init__(self, b_name)  # 親クラスClassBの初期化
        self.name = name
    def print_name(self):
        print("name = {}".format(self.name))
```

python(メタクラス)

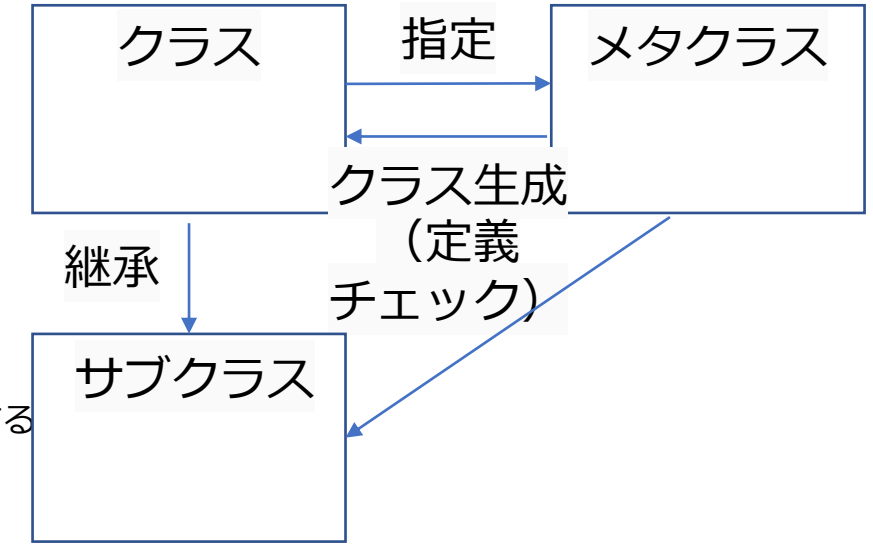
ポリモーフィズムに先立ってメタクラスについて説明します。
メタクラスとは、クラスの再定義をするクラスという風に考えればよいです。
(デフォルトのメタクラスはtypeクラスだが、自身で定義することもできる)

class定義時に、継承と同じ形でmetaclass=メタクラス名とすると自身のメタクラスを指定できます

メタクラスは、主にその定義でよいのかクラスを検証する際に用いられます。

定義は以下のようにします

```
class Meta(type):
    def __new__(metacls, name, bases, class_dict):
        # クラスのチェックを行う
        # name: クラスの名前
        # bases: 継承しているクラス
        # class_dict: クラスの持っている値
        return super().__new__(metacls, name, bases, class_dict) # typeクラスでクラスを生成する
```



python(ポリモーフィズム(多態性))

サブクラスを複数作成し、同じメソッドを定義して呼び出す際にどのクラスか意識せずに呼び出すこと

```
from abc import abstractmethod, ABCMeta

class Human(metaclass=ABCMeta): #親クラス
    def __init__(self, name):
        self.name = name

    @abstractmethod #親クラスでは処理を定義しないメソッド。子クラスで必ずオーバーライドする（オーバーライドしないとABCMetaにはじかれる）
    def say_something(self):
        pass
    def say_name(self): #共通メソッド
        print(self.name)

class HumanA(Human):
    def say_something(self):
        print("I'm A")

class HumanB(Human):
    def say_something(self): # 同じ名前のものを定義
        print("I'm B")
```

python(Private変数)

これまで学んだクラス変数、インスタンス変数は外部からアクセスして値を書き換えられます。
そのため、アクセスの出来ない変数(Private変数を定義する必要があります。)
*) ただし、Pythonは厳密なPrivate変数はなく、以下の方法でも外部からアクセスできる
__variable　**# Private変数 アンダースコア(_)を二つ付けて定義**

```
class Human:
    __msg = "Hello"
    #↑クラス変数のPrivate変数です。

    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        #↑インスタンス変数のPrivate変数です。

    def print_msg(self):
        print("{} name = {}, age = {}".format(self.__msg, self.__name, self.__age)) # クラス内でアクセス

taro = Human("Taro", 20, 'Man')
jiro = Human("Jiro", 18, 'Man')
print(taro.__name)
#↑これだとアクセスできずエラーになります。
print(taro._Human__name)
#↑Private変数ですが、クラス外からも普通にアクセスできます。
```

python(カプセル化, setter, getter)

Private変数を使う場合には、クラスの外部から変数が見えないようにする**(カプセル化)** 必要があります。
カプセル化をする場合には、getterとsetterを定義して利用しないと変数にアクセスができないようにします。

【カプセル化の方法1】
def get_変数名():
def set_変数名():
変数名 = property(get_変数名, set_変数名)

```
class Human:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):    #__nameのgetterの定義
        print("getterを呼び出しました")
        return self.__name        #private変数の__nameを返します。
    def get_age(self):
        return self.__age

    def set_name(self, value):    #__nameのsetterの定義
        print("setterを呼び出しました")
        self.__name = value        # private変数の__nameを設定します。
    def set_age(self, value):
        self.__age = value

    name = property(get_name, set_name) #property関数を利用することで、
get_nameとset_nameを呼び出す機能を持ったnameプロパティが作成されます。
    age = property(get_age, set_age)
```

python(カプセル化, setter, getter)

【カプセル化の方法2】
@property,@var.setter

```
class Human:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property    # @propertyとすると、nameプロパティのgetterとして定義します
    def name(self):
        print("getterを呼び出しました")
        return self.__name        #privateの変数、__nameを返します。
    @property
    def age(self):
        return self.__age

    @name.setter    # nameプロパティのsetterとして定義します。
    def name(self, value):
        print("setterを呼び出しました")
        self.__name = value       # privateの変数、__nameを設定します
    @age.setter
    def age(self, value):
        self.__age = value
```

python(ファイル入力)

ファイルからのテキストの読込みは以下のコマンドを利用する

f = open(file_path, mode='r') **# 読込みモード**
line = f.read() **# ファイルの中身全体を読込み**
f.close() **# ファイルを閉じる**

f = open(file_path, mode='r', encoding='utf-8') **# utf-8でファイルを開く**
f = open(file_path, mode='r', encoding='sjis') **# shift-jisでファイルを開く**
f = open(file_path, mode='r', newline='¥r¥n') **# 改行コードを¥r¥nに指定(crlf)**
lines = f.readlines() **# 行ごとに分割したリストとして取得**
line = f.readline() **# ファイルを一行毎に読込み**

・read, readlinesはファイルを全て一気に読み込むため、処理は速いがメモリの消費は大きい（小さいファイル向け）
・ readlineはファイルを1行ずつ読み込むため、処理は遅いがメモリの消費は小さい（大きいファイル向け）

ファイルのopenとcloseを自動的に行う(with)

```
with open(file_path, mode='r') as f:
    for x in f.readlines():
     ・・・・
```

個with構文を抜けると自動的にファイルがクローズされる

セイウチ演算子(python3.8以降)
ファイルの代入と比較を同時に行う

```
with open('file.csv', mode='r') as f:
    while msg := f.readline():
        print(msg)
```

python(ファイル出力)

ファイル書き込み

f = open(file_path, mode='w') **# 書き込みモード（ファイルが存在しなければ作成、存在した場合は上書き）**
f.write(文字列) **# ファイルに文字列を書き込み（¥nで改行）**
f.close() **# ファイルを閉じる**

f.writelines() **# リストをそのままつなげて書き込み**
f.write('¥n',join(list)) **# リストを改行させて書き込み**

with open(file_path, mode='w', newline='¥r¥n', encoding='sjis') as f:

ファイル追記

f = open(file_path, mode='a') **# 追記モード(ファイルが存在しなければ作成、存在した場合は追記)**

バイナリファイルの読み書き（詳細は割愛）
modeにbを付ける
open(file_path, mode='rb') **#バイナリファイルの読み込み**
open(file_path, mode='wb') **#バイナリファイルの書き込み**

python(with詳細)

withの基本的な使い方について記載します。withは以下のように記述します。

```
with クラス as a:
  処理
```

withの中の処理を実行する前に、withの後に指定したクラスの__init__と__enter__がそれぞれ呼ばれ、処理終了後に、クラスの__exit__メソッドが呼ばれます。

withをどのように使うかというと、処理が連続していてすべての処理が必要な場合、例えば

- ファイルの書き込み処理(ファイル開く→書き込む→ファイル閉じる)
- DBへのデータの書き込み処理(DBへコネクションを張る→書き込む→コネクションを閉じる)

```
class WithTest(object):
  def __init__(self):
    print("initが呼ばれました")
  def __enter__(self):
    print("enterが呼ばれました")
  def __exit__(self, exc_type, exc_val, exc_tb):
    print("exitが呼ばれました")
with WithTest() as t:
  print("withを実行しました")
  #个withでクラスを作成すると、まず__init__を実行、次に__enter__を実行、処理終了後に、__exit__を実行します。
  # 以下の順で実行されます。
  # initが呼ばれました
  # enterが呼ばれました
  # withを実行しました
  # exitが呼ばれました
```

基本講座３(まとめ)

クラスの定義・・プロパティ、クラスのメソッド、インスタンスの作成

クラス変数とインスタンス変数

コンストラクタ(__init__)、デストラクタ(__del__)

インスタンスメソッド、クラスメソッド、スタティックメソッド

特殊メソッド

オブジェクト指向・・・継承、ポリモーフィズム、カプセル化(Private変数、 setter, getter)

クラスの多重継承、メタクラス

ファイル入出力、with