Python 入門講座 第 11 回

February 9, 2022

1 Python 入門講座　第 11 回: 名前って何？　What's in a name?

プログラムを作成する際には、変数や関数、クラス、モジュールなど様々な名前が現れます。これらの名前はその有効範囲 (スコープ) の中では一意的に定まっている必要があります。

名前の有効範囲を把握しておかないと、プログラムが意図しない動作をする場合も考えられます。(プログラムの違う場所で、同じ名前を違う意味／目的で使っていると、思わぬ動作をすることがあります。)

python プログラム中で導入された**名前**の有効範囲は一定のルールで定まっています。ある時点での有効な名前の集合（辞書）を名前空間と呼んだりします。(実際には、名前空間とスコープは区別なく使われることがあるかもしれません。)

python ではモジュール (プログラムファイル)、関数定義、クラス定義によって新たな名前空間が作られます。

1.1 名前のスコープの例（関数定義）

いま名前のスコープの例として、二つの数の和を返す関数を考えて見ます。

[1]:
```
import logging
from logging import info,warning, getLogger, INFO
getLogger().setLevel(INFO)

def add2(x:int, y:int = 1) -> int:
    return x+y
```

これを使って、次の python プログラムを実行したとします。

```
x=10; y=20 ; Y = 100
print(f"global {add2(3)=} , {x=}, {y=}")
```

add2(3) の値は　10 + 20 = 30 or 10 + 1 = 11 or 3 + 1 = 4 or 3 + 20 = 5 のいずれになるでしょうか？

実際に試して見ましょう。

[2]:
```
def add2(x:int, y:int = 1)->int:
    return x+y

x = 10; y = 20 ; Y = 100
logging.warning(f" {add2(3) = } , {x = }, {y = }")
```

```
WARNING:root: add2(3) = 4 , x = 10, y = 20
```

結果は、

```
add2(3) = 4 , x = 10, y = 20
```

となります。関数定義の中の x や y は関数定義の外で定義された x や y とは無関係ということがわかります。このことを、関数定義の中の y と関数定義の外で定義された y は「異なるスコープを持つ」、あるいは「別の名前空間に属している」といいます。

python では、関数定義の中の y は関数 add2 の local な名前空間に属しています。一方, 関数定義の外で定義された y はこのプログラムの global な名前空間に属しています。

さて、関数定義の中で x + y を x + Y と書き間違えた時、なにが起きるかを見てみましょう。

[3]:
```
from logging import info,warning, getLogger, INFO
getLogger().setLevel(INFO)

def add2(x:int, y:int = 1)->int:
    Y=200
    info(f"in add2: {x=:}, {y=:}, {Y=:}")
    return x + Y # もし　x + Yと書き間違えたとしたら？

x=10;y=20;Y = 100
info(f"global {add2(3) = } {x = } {y = } {Y=} ")
```

```
INFO:root:in add2: x=3, y=1, Y=200
INFO:root:global add2(3) = 203 x = 10 y = 20 Y=100
```

このように、名前の有効範囲を正しく把握しておかないと、プログラムは動作するけれど、意図した正しい結果を返さない場合もあるということです。

注意：Python には C/C++などのコンパイル言語と違って、変数名の宣言文はありません。そのため、実行時に意図せず同じ変数名を異なる意味で使った時にも、エラーにならない場合があります。pylint や flake8 などのツールを使うことで、プログラム実行前にこれらのエラーの可能性をチェックできます。

1.2 新しい名前が現れる場所

先ほどは、関数定義のブロックの中に現れる変数名を例に、名前空間について説明しました。

python で新しい名前が現れる (名前空間に新しい名前が追加される）場所は、次のようなものがあります。

- 代入が行われるときの代入対象の識別子：代入文　x=1 の x, 代入式　v:=2 の v　など
- クラスや関数の定義：def func(): の func ,class　c:の c
- 関数の仮引数：def func(x) の x, lambda z:print(z) の z
- for ループのヘッダ: for i in (1,2,3) の i
- with 文や except 節の as の後ろ:with open("file1") as fin の fin や except UnboundLocalError as err の err
- import 文：import myModule の myModule , impolrt myModule as mm の mm
- from … import 文：　from mod import component の component, from mod import component as comp の comp

– `from ... import *`では import されるモジュール内で定義されている, アンダースコアから始まるもの以外の全ての名前を束縛します。

これらの場所では、新しい名前にオブジェクトを結びつけています (束縛). 新しい名前が束縛されたときその名前はその時の名前空間 (local 名前空間) に登録されます。

モジュール (~一つの python プログラムファイル)、関数定義、クラス定義は新しい名前空間を作成します。

`global` 名前空間はモジュールの名前空間と組み込み名前空間 (`__builtins__`の名前空間) を合わせた名前空間です。組み込み名前空間には、python が標準的に提供する関数、定数、クラスなどの名前が登録されています。

`__builtins__`名前空間に登録されている名前は、

`dir(__builtins__)`

で確認することができます。

余談：

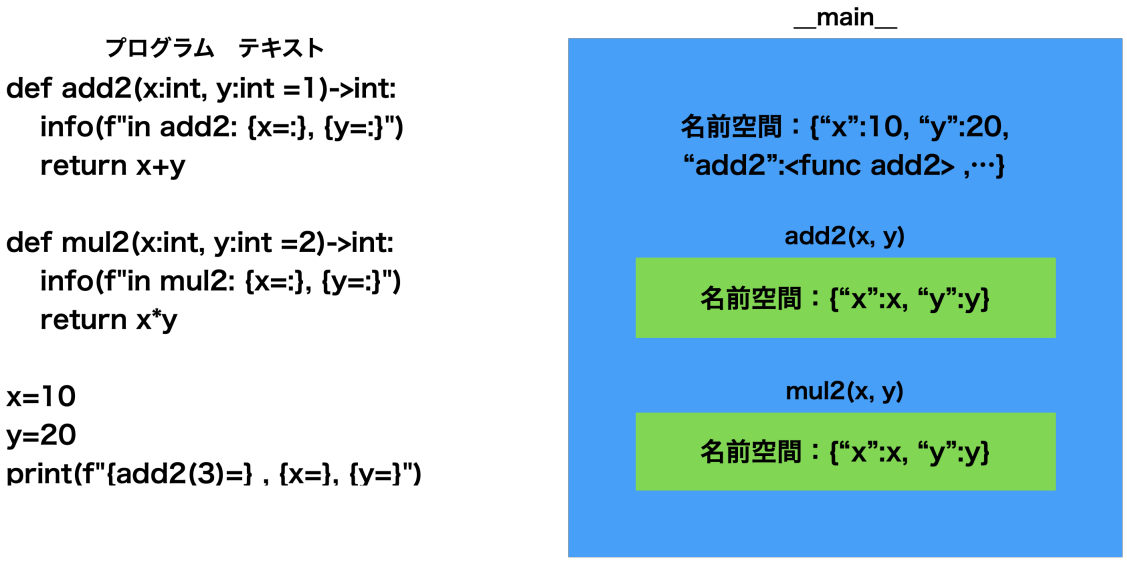
Python プログラムにはシステムがあらかじめ定義している語として、`def`,`if` などプログラム言語としての文法要素である予約語と python があらかじめ用意している定数や関数などの組み込みオブジェクト名があります。　予約語は文法規則で決まっているため変更することができませんが、組み込みの変数や関数の名前は、他の名前 (識別子, identifier) と同じように、別のオブジェクトに割り当てる (束縛する) こともできます。　しかし、組み込みオブジェクトの束縛を変更することは理解しにくいプログラムを作る元になりますので、避けましょう。(`True`, `False`, `None` の三つの定数は予約後であると同時に、`__builtins__` 名前空間にも登録されています。)

[4]:
```
x=1
(lambda x:logging.warning(f"lambda {x = }, {locals() = }"))(10)
logging.warning(f"global {x = }")

exec('(lambda x:logging.warning(f"exec_lambda {x = }, {locals() =␣
 ↪}"))(20)\nlogging.warning(f"exec_global {x = }")')

[w for i in range(10) if (w:=i)%3 == 0]
logging.warning(f"global {w = }")
```

```
WARNING:root:lambda x = 10, locals() = {'x': 10}
WARNING:root:global x = 1
WARNING:root:exec_lambda x = 20, locals() = {'x': 20}
WARNING:root:exec_global x = 1
WARNING:root:global w = 9
```

1.2.1　名前の解決

ある名前がプログラム中で使われる時、この名前に結び付けられている (束縛されている) オブジェクトを見つける必要があります。Python では `local()` の名前空間から出発し、`global()` までの階層的な名前空間を 1 段階ずつ登りながら検索します。`globals()` にもこの名前がみつからなければ、`Error` が生成されます。



関数定義、クラス定義などのブロックはそれぞれの名前空間を持っています。一つのファイルに含まれるプログラムテキストはグローバルな名前空間を定義しています。プログラム中の名前はこの階層的な構造を持つ名前空間を local な名前空間からグローバル空間まで一段ずつ検索され、束縛されているオブジェクトを見つけます。

python では、関数定義のブロック、クラス定義のブロックの内部にクラス定義や関数定義を置けます。” 関数プログラミング” のスタイルではこのような階層構造を反映して、名前空間も global,locals

1.3 dir() 関数

dir() 関数は、引数なしで実行された時、実行時のスコープに含まれる名前のリストを返します。

[5]:
```
print(dir())
```

```
['INFO', 'In', 'Out', 'Y', '_', '__', '___', '__builtin__', '__builtins__',
'__doc__', '__loader__', '__name__', '__package__', '__spec__', '_dh', '_i',
'_i1', '_i2', '_i3', '_i4', '_i5', '_ih', '_ii', '_iii', '_oh', 'add2', 'exit',
'getLogger', 'get_ipython', 'info', 'logging', 'quit', 'w', 'warning', 'x', 'y']
```

1.3.1 globals() 関数と locals() 関数

プログラムのある時点での global なスコープを持つ名前の辞書 (名前空間) を global() 関数呼び出しで、また local なスコープを持つ名前を local() 関数呼び出しで確認できます。例をみて見ましょう。

[6]:
```
def add2(x:int, y:int =1)->int:
    info (f" in {__name__}: {dir()} {locals()=}")
    info (f"local  scope:{[e for e in locals().items()]}")
    info (f"global scope:{[(k,v) for k,v in globals().items() if k in␣
 ↪('x','y')]}")
    return x+y
x=1
```

```
y=2
print(add2(3), x, y)
y=4
print(add2(3), x, y)
```

```
INFO:root: in __main__: ['x', 'y'] locals()={'x': 3, 'y': 1}
INFO:root:local  scope:[('x', 3), ('y', 1)]
INFO:root:global scope:[('x', 1), ('y', 2)]
INFO:root: in __main__: ['x', 'y'] locals()={'x': 3, 'y': 1}
INFO:root:local  scope:[('x', 3), ('y', 1)]
INFO:root:global scope:[('x', 1), ('y', 4)]

4 1 2
4 1 4
```

というように、add2(3) を実行している時の local スコープでは ('x', 3), ('y', 1) などとなっています。計算にはこれらの local スコープの値が使われ、global スコープの値は影響を与えていません。

この関数定義で仮引数 y の記述を忘れて、

```
def add2(x):
    info (f"local  scope:{[e for e in locals().items()]}")
    info (f"global scope:{[(k,v) for k,v in globals().items() if k in ('x','y')]}")
    return x+y
```

としてしまった場合を考えて見ます。

[7]:
```
def add2(x):
    info (f"local  scope:{[e for e in locals().items()]}")
    info (f"global scope:{[(k,v) for k,v in globals().items() if k in␣
 ↪('x','y')]}")
    return x+y
x=1;y=2
print(add2(3), x, y)
x=5;y=6
print(add2(3), x, y)
```

```
INFO:root:local  scope:[('x', 3)]
INFO:root:global scope:[('x', 1), ('y', 2)]
INFO:root:local  scope:[('x', 3)]
INFO:root:global scope:[('x', 5), ('y', 6)]

5 1 2
9 5 6
```

この場合には、関数 add2(3) を実行する際の local 名前空間には変数 y がありません。python は名前空間をたどり、global 空間にある名前 y を見つけて、その値を使って関数の値を計算します。

この例のように、関数の値が関数定義の外の変数の影響を受けて変わってしまうと、関数の動作の確認／検証を難しくしてしまいます。　一般的にはこのような関数の定義は避けるべきものとされています。しかしながら、問題によっては、関数外にシステムを特徴付ける数値 (~定数) があって、それ

に基づいて処理を定義することを求められることがあります。このような場合にも、global 宣言を使って関数定義外の変数を使っていることがを明確する必要があります。

[8]:
```
def sumg(x):
    global y
    y=10
    info (f"local  scope:{[e for e in locals().items()]}")
    info (f"global scope:{[(k,v) for k,v in globals().items() if k in␣
 ↪('x','y')]}")
    return x+y
x=1
y=2
print(x,y, sumg(3), x, y)
y=4
print(x,y, sumg(3), x, y)
```

```
INFO:root:local  scope:[('x', 3)]
INFO:root:global scope:[('x', 1), ('y', 10)]
INFO:root:local  scope:[('x', 3)]
INFO:root:global scope:[('x', 1), ('y', 10)]
```

```
1 2 13 1 10
1 4 13 1 10
```

[9]:
```
def incy(z):
    y=z
    info (f"local  scope:{[e for e in locals().items()]}")
    def func(x):
        nonlocal z
        info (f"local  scope:{[e for e in locals().items()]}")
        info (f"global scope:{[(k,v) for k,v in globals().items() if k in␣
 ↪('x','y','z')]}")
        return (x+y)
    return func
x=10; y=20; z=30
print(incy(4)(3), x, y, z)
print(incy(10)(20), x, y, z)
```

```
INFO:root:local  scope:[('y', 4), ('z', 4)]
INFO:root:local  scope:[('x', 3), ('y', 4), ('z', 4)]
INFO:root:global scope:[('x', 10), ('y', 20), ('z', 30)]
INFO:root:local  scope:[('y', 10), ('z', 10)]
INFO:root:local  scope:[('x', 20), ('y', 10), ('z', 10)]
INFO:root:global scope:[('x', 10), ('y', 20), ('z', 30)]
```

```
7 10 20 30
30 10 20 30
```

1.3.2 global 宣言と nonlocal 宣言

Python の名前の検索ルールから関数やクラスの中から一つ上のレベルの名前空間の変数に束縛されたオブジェクトを入手することがで来ます。しかし、この場合には内側の名前空間を持つブロックでは、このオブジェクトを変更することができません。この場合、global 宣言や nonlocal 宣言を使うことで、内側の名前空間からこれらの変数に束縛されたオブジェクトを入手できます。global,nolocal で宣言される変数は、宣言の文が実行される前にそれらの名前空間で定義（束縛）されている必要があります。

[10]:
```
# global と local
import logging

x=1

def foo(): # 代入文の左辺に現れない変数は global として取り扱われる。
    logging.warning(f"{x =}")
    return x

logging.warning(f"{x = }, {foo() = } , {x = }")

def bah(): #代入文の左辺に現れる
    x=10
    return x

def gee(): # 代入文の左辺に現れる変数を、代入前にその値を使おうとしている。
    #print(globals())
    try:
        x=x+1 #
        return x
    except UnboundLocalError as err:
        logging.warning(f"{__name__} {err}")
        return None

def huhu(): # global 宣言すると、その変数名は global 空間の変数として扱われる。
    global x
    try:
        x=x+1 #
        return x
    except UnboundLocalError as err:
        logging.warning(err)
        return None

logging.warning(f"{x = }, {foo() = } , {x = }")

logging.warning(f"{x = },{bah() =} ,{x = }")

logging.warning(f"{x = }, {gee() = }, {x = }")
```

```
logging.warning(f"{x = }, {huhu() = }, {x = }")
```

```
WARNING:root:x =1
WARNING:root:x = 1, foo() = 1 , x = 1
WARNING:root:x =1
WARNING:root:x = 1, foo() = 1 , x = 1
WARNING:root:x = 1,bah() =10 ,x = 1
WARNING:root:__main__ local variable 'x' referenced before assignment
WARNING:root:x = 1, gee() = None, x = 1
WARNING:root:x = 1, huhu() = 2, x = 2
```

[11]:
```
def gen_add_n(n):
    def add_n(x):
        nonlocal nl
        nl +=1
        print(f"{n = }, {nl = }",end=None)
        return (x+n)
    nl=0
    return add_n

add_1=gen_add_n(1)
add_2=gen_add_n(2)

print(f"{add_1(10) =}, {add_2(10) =}")
print(f"{add_1(20) =}, {add_2(20) =}")
if "n" in locals():print(n)
```

```
n = 1, nl = 1
n = 2, nl = 1
add_1(10) =11, add_2(10) =12
n = 1, nl = 2
n = 2, nl = 2
add_1(20) =21, add_2(20) =22
```

1.4 名前の付け方 (PEP 8 から)

Python プログラムで使う名前では、ここまでで説明したように、名前の有効範囲に注意を払う必要があります。名前付の際に一定のルールを採用することで、名前の衝突をさけ、プログラムの予期せぬ動作を避けることも推奨されています。

python で使われる名前についてのガイドラインが PEP(Python Enhancement Proposals) に示されています。完全にこれに従う必要はありませんが、見やすい Python プログラムを作るための参考になるでしょう (PEP8 の第 2 セクションもご覧ください)。

多くの方に使われるであろうモジュールを公開する際には、この命名規則に準拠することが推奨されます。`pylint` などの python プログラム診断ツールを使って、この命名規則に対するチェックを行えます。

[12]:
```
help(print)
```

```
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.
```

1.4.1 実践されている命名方法の例

PEP8 の「命名規則」の中で列挙されている名前付の方法の例を見て見ましょう。

- b (小文字 1 文字)
- B (大文字 1 文字)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (CapWords, または **CamelCase** - 文字がデコボコに見えることからこう呼ばれます)。StudlyCaps という呼び名でも知られています。
  - 注意: CapWords の中で頭字語 (大文字だけで綴られる言葉。NATO、HTTP など) を含める場合、頭字語の全ての文字を大文字にします。つまりこのやり方だと、HttpServerError より HTTPServerError の方が良いということになります。
- mixedCase (はじめの文字が小文字である点が、CapitalizedWords と違います！)
- Capitalized_Words_With_Underscores (醜い！)

関連する名前の集まりに、短い一意なプレフィックスを付けるやり方もあります。Python ではこのやり方を多く使っているわけではありませんが、完全を期すために紹介しておきます。

python プログラムの慣習として"_"(アンダースコア) で始まる (`_single_leading_underscore`) あるいは終わる (`single_trailing_underscore_`) 名前には次のような意味が含まれています。

"_"(アンダースコア) で始まる, あるいは終わる名前

1. 名前が一つの"_"で始まる名前 (`_single_leading_underscore`) : 内部的に使われる名前 (`non-public`) を意味します。module を `from ... import *`構文で `import` する際には、これらの名前は `import` されません。
2. 名前が二つの"_"で始まる名前 (`__double_leading_underscore`): クラスの属性名として使われた時、クラス内でのみ有効な属性名であることを意味します。(名前のマングリング機構によってクラス外からはこの名前ではアクセスすることはできません。)
3. 名前が一つの"_"で終わる名前 (`single_trailing_underscore_`): python のキーワードとの衝突を避けるために使われます。例: `tkinter.Toplevel(master, class_='ClassName')`
4. 名前が二つの"_"で始まり、二つの"_"で終わる名前 (`__double_leading_and_trailing_underscore__`): python がシステムで持っている" マジック" オブジェクト または "マジック" 属性です。この

種類の名前を新たに定義することは、将来問題を引き起こす可能性がありますので、避けなければなりません。

1.5 守るべき命名規約

PEP 8 が述べている「守るべき命名規約」を簡単にまとめておきます (一部を省略しています)。

1.5.1 こんな名前は嫌だ

単一の文字 'l' (小文字のエル)、'O' (大文字のオー)、'I'(大文字のアイ) を決して変数に使わないでください。

フォントによっては、これらの文字は数字の 1 や 0 と区別が付かない場合があります。'l'(小文字のエル) を使いたくなったら、'L' を代わりに使いましょう。

1.5.2 ASCII との互換性

標準ライブラリ で使われる識別子は、ASCII と互換性がなければなりません (PEP 3131)。

1.5.3 パッケージとモジュールの名前

モジュールの名前は、全て小文字の短い名前にすべきです。読みやすくなるなら、アンダースコアをモジュール名に使っても構いません。

Python のパッケージ名は、全て小文字の短い名前を使うべきですが、アンダースコアを使うのは推奨されません。(パッケージとは、複数のモジュールを階層的にまとめたものを、あたかも一つのモジュールのように取り扱う仕組みです。)

1.5.4 クラスの名前

クラスの名前には通常 CapWords 方式を使うべきです。

主に callable として使われる、ドキュメント化されたインターフェイスの場合は、クラスではなく関数向けの命名規約を使っても構いません。

Python にビルドインされている名前には別の規約があることに注意してください : ビルトインされている名前のほとんどは、単一の単語 (または、二つの単語が混ざったもの) ですが、例外的に CapWords 方式が使われている名前や定数も存在しています。

1.5.5 例外の名前

例外はクラスであるべきです。よって、クラスの命名規約がここにも適用されます。しかし、その例外が実際にエラーである場合には例外の名前の最後に "Error" をつけるべきです。

1.5.6 関数や変数の名前

関数の名前は小文字のみにすべきです。また、読みやすくするために、必要に応じて単語をアンダースコアで区切るべきです。

変数の名前についても、関数と同じ規約に従います。

mixedCase が既に使われている (例: threading.py) 場合にのみ、互換性を保つために mixedCase を許可します。

1.5.7 グローバル変数の名前

(ここで言う「グローバル変数」はモジュールレベルでグローバルという意味だと思いたいですが) ここで示す規約は、関数レベルのものと同じです。

`from M import *` 方式で import されるように設計されているモジュールは、グローバル変数をエクスポートするのを防ぐため **all** の仕組みを使うか、エクスポートしたくないグローバル変数の頭にアンダースコアをつける古い規約を使うべきです (こうすることで、これらのグローバル変数は「モジュールレベルで公開されていない」ことを開発者が示したいかもしれません)。

(モジュールのグローバル変数`__all__`に export する名前 (`str`) のリストを設定することで、`from ... import *`で import される名前を指定しておくことができます。)

1.6 定数 の名前

定数は通常モジュールレベルで定義します。全ての定数は大文字で書き、単語をアンダースコアで区切ります。例として `MAX_OVERFLOW` や `TOTAL` があります。

1.6.1 関数やメソッドに渡す引数

- インスタンスメソッドのはじめの引数の名前は常に `self` を使ってください。
- クラスメソッドのはじめの引数の名前は常に `cls` を使ってください。
- 関数の引数名が予約語と衝突していた場合、アンダースコアを引数名の後ろに追加するのが一般的には望ましいです。衝突した名前を変更しようとして、略語を使ったりスペルミスをするよりマシです。よって、`class_` は `clss` より好ましいです。(多分、同義語を使って衝突を避けるのがよいのでしょうけど)

1.6.2 メソッド名とインスタンス変数

- 関数の命名規約を使ってください。つまり、名前は小文字のみにして、読みやすくするために必要に応じて単語をアンダースコアで区切ります。
- **公開されていない**メソッドやインスタンス変数にだけ、アンダースコアを先頭に付けてください。
- サブクラスと名前が衝突した場合は、Python のマングリング機構を呼び出すためにアンダースコアを先頭に二つ付けてください。
  - Python はアンダースコアが先頭に二つ付いた名前にクラス名を追加します (マングリング機構)。つまり、クラス `Foo` に `__a` という名前の属性があった場合、この名前は `Foo.__a` ではアクセスできません (どうしてもアクセスしたいユーザーは `Foo._Foo__a` とすればアクセスできます)。一般的には、アンダースコアを名前の先頭に二つ付けるやり方は、サブクラス化されるように設計されたクラスの属性が衝突したときに、それを避けるためだけに使うべきです。