

Python 型のある世界へ

typingモジュール紹介



背景

なぜPythonに型をつけたいのか？

PyCharm 2020.2

```
1 def move(point, delta_x, delta_y):
2     point.x += delta_x
3     point.|
```

f x

- if
- ifn
- ifnn
- len
- main
- not
- par
- print
- return
- while

Ctrl+下 and Ctrl+上 will move caret down and up in the editor [Next Tip](#)

pointがxを持っていない場合
エラーになるのでは？

型を定義していないのに、
なぜ勝手に「x」が入力候補に出ているの？

なぜここで使わない選択肢を
沢山推薦してくれるの？
(実はSnippetです)

背景

なぜPythonに型をつけたいのか？

PyCharm 2020.2

```
1 class Point:
2     def __init__(self, x = 0, y = 0):
3         self.x: float = x
4         self.y: float = y
5
6 def move(point: Point, delta_x, delta_y):
7     point.x += delta_x
8     point.
```

f x
f y
m __init__(self, x, y)
f __doc__
f __module__
f __annotations__
p __class__

Point
Point
Point
Point
Point
object
object

型を指定してみれば？

明確に定義した「x」と「y」が
前に挙げられた！

objectのbuiltinメソッドも
2番目に挙げられた。

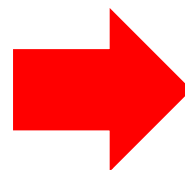
型をつけるメリット？

Pythonでは型をつけなくても動くが、型をつけると以下のメリットがある。

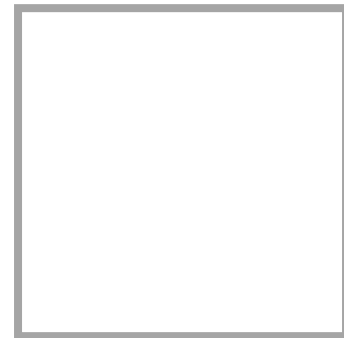
- ◆「変数がどんな型か」 がコード編集時にはっきりわかる。
- ◆「オブジェクトがどんなプロパティ、メソッドを持っているのか」 が入力候補リストから分かる。
- ◆「methodや変数に違う型の値を渡していないか」 がコード編集時すぐに分かる 。



- ◆ IDEが入力候補をより正確に出せる。
- ◆ コードを実行しなくても過ちを早く発見できる。

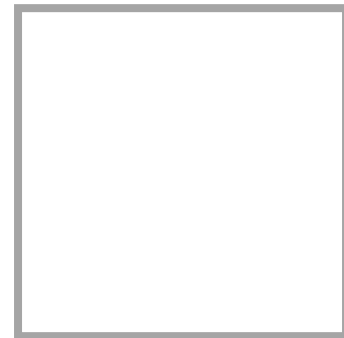


コードの規模が膨大であるほど、
差がつく。



Pythonにおける型チェックの特徴

- ◆ 強制力はない
 - ◆ 型の指定はStatic Type Checkerのためのものであり、実行段階では一切無視される。
- ◆ Static Type Checkerの実装によって、チェックの完成度も違う
 - ◆ 使用するIDEやSTC Pluginの違いによって、チェックの完成度が違う。
(あくまで個人的な感想ですが、PyCharmよりVSCode + Pylance(Pyright)の方が完成度高い)
- ◆ 高度な型チェックを実現するためにビルトインの「typing」モジュールを使う



型 を明確にしよう

typingモジュールの使い方



変数の型

変数の型を指定

{変数名} : {型}

```
5  #変数の型チェック-----
6  a: int
7  a = 1  #変数型チェック : OK
8  a = 1.0  #変数型チェック : 警告
9  b: float = "1"  #変数型チェック : 警告
10
11
12  def func(x: float) -> float: #警告 : returnなしの可能性もある。PyCharmではチェックされない !
13      if x > 1:
14          return 1  #返値型チェック : OK
15      elif x > 2:
16          return 1.0  #返値型チェック : OK
17      elif x > 3:
18          return "a"  #返値型チェック : 警告
19
20
21  a = func(1)  #引数型チェック : OK, a変数型チェック : 警告
22  b = func(1.0)  #引数型チェック : OK, b変数型チェック : OK
23  b = func("1")  #引数型チェック : 警告
```


typing.List & typing.Dict

リストと辞書にイれるオブジェクトの型を指定

List[{型}], Dict[{key型}, {value型}]

```

26 #配列内容の型を与える-----
27 from typing import List, Dict
28
29
30 class A:
31     pass
32
33
34 class B1(A):
35     pass
36
37
38 class B2(A):
39     pass
40
41
42 class C1(B1):
43     pass

```

```

47 c: List[B1]
48 c = [B1()] #配列要素型チェック : OK
49 c = [C1()] #配列要素型チェック : OK
50 c = [A()] #配列要素型チェック : 警告
51 c = [B2()] #配列要素型チェック : 警告
52 c = [A(), B1()] #配列要素型チェック : 警告
53 c.remove(A()) #引数型チェック : OK
54 c.remove(B1()) #引数型チェック : OK
55 c.remove(B2()) #引数型チェック : 警告
56 c.remove(C1()) #引数型チェック : OK
57
58 d: Dict[str, B1] = dict()
59 d["b"] = A() #KeyとValue型チェック : 警告
60 d["b"] = B1() #KeyとValue型チェック : OK
61 d["b"] = B2() #KeyとValue型チェック : 警告
62 d["b"] = C1() #KeyとValue型チェック : OK
63 d[1] = B1() #KeyとValue型チェック : 警告

```

indexingではなくgenericsです !
C#, Javaの「<>」に相当

C# :
`new List<B1>()`

Java:
`new ArrayList<B1>()`

typing.List & typing.Dict

リストと辞書の中身の型を指定

List[{型}]. Dict[{key型}, {value型}]

```

26 #配列内容の型を与える-----
27 from typing import List, Dict
28
29
30 class A:
31     pass
32
33
34 class B1(A):
35     pass
36
37
38 class B2(A):
39     pass
40
41
42 class C1(B1):
43     pass

```

```

47 c: List[B1]
48 c = [B1()] #配列要素型チェック : OK
49 c = [C1()] #配列要素型チェック : OK
50 c = [A()] #配列要素型チェック : 警告
51 c = [B2()] #配列要素型チェック : 警告
52 c = [A(), B1()] #配列要素型チェック : 警告, PyCharmではB1要素が含まんでいればOKになってしまう。
53 c.remove(A()) #引数型チェック : 警告
54 c.remove(B1()) #引数型チェック : OK
55 c.remove(B2()) #引数型チェック : 警告
56 c.remove(C1()) #引数型チェック : OK
57
58 d: Dict[str, B1] = dict()
59 d["b"] = A() #KeyとValue型チェック : 警告
60 d["b"] = B1() #KeyとValue型チェック : OK
61 d["b"] = B2() #KeyとValue型チェック : 警告
62 d["b"] = C1() #KeyとValue型チェック : OK
63 d[1] = B1() #KeyとValue型チェック : 警告

```

typing.Generic

ユーザー定義Generic

Generic[{TypeVar変数}, ...]

```

1  class A:
2      pass
3
4
5  class B1(A):
6      pass
7
8
9  class B2(A):
10     pass
11
12
13 class C1(B1):
14     pass
15

```

```

16 #ユーザー定義のジェネリック型-----
17 from typing import TypeVar, Generic, Union
18
19 T1 = TypeVar("T1", B1, B2) #B1とB2を受け入れる。サブクラスを含まない。
20 T2 = TypeVar("T2", bound=Union[B1, B2]) #B1とB2を受け入れる。サブクラスを含む。
21 T3 = TypeVar("T3") #任意の型を受け入れる
22
23 class MyList_T1(Generic[T1]):
24     ...
25
26 class MyList_T2(Generic[T2]):
27     ...
28
29 class MyList_T3(Generic[T3]):
30     ...
31
32 class MyDict_T3_T2(Generic[T3, T1]):
33     ...

```

```

36 myList1 = MyList_T1[A]() #Genericタイプチェック : 警告
37 myList1 = MyList_T1[B1]() #Genericタイプチェック : OK
38 myList1 = MyList_T1[B2]() #Genericタイプチェック : OK
39 myList1 = MyList_T1[C1]() #Genericタイプチェック : 警告
40
41 myList2 = MyList_T2[A]() #Genericタイプチェック : 警告
42 myList2 = MyList_T2[B1]() #Genericタイプチェック : OK
43 myList2 = MyList_T2[B2]() #Genericタイプチェック : OK
44 myList2 = MyList_T2[C1]() #Genericタイプチェック : OK
45
46 myList3 = MyList_T3[A]() #Genericタイプチェック : OK
47 myList3 = MyList_T3[B1]() #Genericタイプチェック : OK
48 myList3 = MyList_T3[B2]() #Genericタイプチェック : OK
49 myList3 = MyList_T3[C1]() #Genericタイプチェック : OK

```

typing.Union

複数の型をOR条件で指定する。

Union[{型}, ...]

```
1  class A:
2      pass
3
4  class B:
5      pass
6
7  class B_sub(B):
8      pass
9
10 class C():
11     pass
12
```

```
14  #複数の型を許可する型を作る-----
15  #C#のGenericsより高い自由度を持ち、
16  #無関係の複数型を許可できる。
17  #(C#では同じインターフェースまたは同じSuper Classの派生クラスである必要がある)
18  from typing import Union, List
19
20  e: List[Union[A, B]]
21  e = [A()]      # 内容物型チェック : OK
22  e = [B()]      # 内容物型チェック : OK
23  e = [B_sub()]  # 内容物型チェック : OK
24  e = [C()]      # 内容物型チェック : 警告
25  e.append(C())  # 引数型チェック : 警告
```


typing.Callable

methodの形式を指定(C#で言うとdelegate)

Callable[[{引数型}, ...], {返値型}]

```

1 class A:
2     pass
3
4
5 class Sub_A(A):
6     pass
7
8 #メソッドの形式を指定する-----
9 from typing import Callable, Any
10
11
12 def func1(x: A) -> float:
13     pass #PyCharmではreturnなしの場合はチェックされない
14
15 def func2(x: Sub_A) -> float:
16     pass
17
18 def func3(x: A) -> None:
19     pass
20
21 def func4(x: int) -> None:
22     pass
23
24 def func5() -> float:
25     pass

```

```

27 h: Callable[[A], float]
28 h = func1 #メソッドの形式チェック: OK
29 h = func2 #メソッドの形式チェック: 警告。引数のタイプが違う。サブクラスは含まない。
30 h = func3 #メソッドの形式チェック: 警告。リターンのタイプが違う。
31 h = func4 #メソッドの形式チェック: 警告。引数のタイプが違う。
32 h = func5 #メソッドの形式チェック: 警告。引数数が違う。
33 # Pycharmでは引数欠けている場合はOKとされてしまう。
34
35 h2: Callable[[Any], float] # リターンの型だけ制限したいとき
36 h2 = func1 #メソッドの形式チェック: OK
37 h2 = func2 #メソッドの形式チェック: OK
38 h2 = func3 #メソッドの形式チェック: 警告。リターンのタイプが違う。
39 h2 = func4 #メソッドの形式チェック: 警告。リターンのタイプが違う。
40 h2 = func5 #メソッドの形式チェック: 警告。引数数が違う。
41
42 h3: Callable[..., float] # リターンの型だけ制限したいとき、引数の数も制限しない。
43 h3 = func1 #メソッドの形式チェック: OK
44 h3 = func2 #メソッドの形式チェック: OK
45 h3 = func3 #メソッドの形式チェック: 警告。リターンのタイプが違う。
46 h3 = func4 #メソッドの形式チェック: 警告。リターンのタイプが違う。
47 h3 = func5 #メソッドの形式チェック: OK
48
49 # 「...」はEllipsisと言う。

```

typing.Callable

応用シーン → decoratorの適用対象を制限したい。

```
51 #メソッドの形式を指定するーその2 : Decoratorの適用範囲を制限する-----
52 from typing import Callable
53
54 method_format = Callable[[int], None]
55
56 def print_name_when_runned(method: method_format) -> method_format:
57     def wrapper(x: int) -> None:
58         #元のメソッドを実行した上で、何らかの手を加えるのがdecoratorの目的。
59         print(f"{method.__name__}が実行された!")
60         method(x)
61     return wrapper
62
63 @print_name_when_runned # Decorator適用対象メソッドの形式チェック : 警告
64 def method1():
65     pass
66
67 @print_name_when_runned # Decorator適用対象メソッドの形式チェック : OK
68 def method2(x: int):
69     pass
70
71 method2(1) # Consoleに「method2が実行された!」と出力
```

typing.overload

メソッドの呼び出し方を簡単に宣告する。

```
1  #overload-----
2  from typing import overload, Union
3
4  class SomeClass():
5      @overload
6      def method1(self, x: int) -> None: #関数可能な呼び出し方を定義のみ、実装できない。
7          ...
8
9      @overload
10     def method1(self, x: str) -> None: #関数可能な呼び出し方を定義のみ、実装できない。
11         ...
12
13     @overload
14     def method1(self, x: float, y: int) -> None: #関数可能な呼び出し方を定義のみ、実装できない。
15         ...
16
17     def method1(self, x: Union[int, str, float], *y: int) -> None: #本番の実装。上記のoverloadで型チェックする
18         if isinstance(x, int):
19             print("int")
20         elif isinstance(x, str):
21             print("str")
22         elif len(y) > 0: #ここでx,とyの型をチェックすると逆に余計なチェックと見なされ、警告が出されます。
23             print("float, int")
24         else:
25             raise TypeError("処理できない型!")
```

```
27  instance = SomeClass()
28  instance.method1(1) #呼出型チェック : OK
29  instance.method1("1") #呼出型チェック : OK
30  instance.method1(1.0, 1) #呼出型チェック : OK
31  instance.method1(1.0) #呼出型チェック : 警告
```


typing.cast

変数型が不明の時、自分で型を指定します。

```
1  #cast-----
2  from typing import cast
3  def cast_test_method(x: int) -> None:
4      print(x + 1)
5
6  cast_test_method(cast(int, 1.0)) # 「2.0」と出力
7  cast_test_method(cast(int, "1")) # エラー
```

typing.Final

変数を上書きできないようにする。

```
1  #final-----
2  from typing import final #python 3.8から
3  from typing import Final #python 3.9から
4
5  class final_test_class():
6      X_var : int = 1 # 一般変数
7      X_final : Final[int] = 1 # Final変数
8      Y_final : Final[int] # Final変数。後で変更できないので、初期値が必須
9
10     @final
11     def method1(self) -> None:
12         self.Y_final = 2
13         ...
14
15     class fsub_inal_test_class(final_test_class):
16         X_var = 2 # 上書きチェック : OK
17         X_final = 2 # 上書きチェック : 警告
18         def method1(self) -> None: # 上書きチェック : 警告
19             ...
```

typing.Type

変数またはGenericsの対象がClassである場合

やりたい事

Pizzaのサブクラスだけ入れたい！

```
menu = { "ピザ名": ピザのクラス }
```

```
def order_pizza(pizza_name: str) -> Pizza:  
    pizza_class = menu[pizza_name]  
    new_pizza = pizza_class()  
    return new_pizza
```


typing.Type

変数またはGenericsの対象がClassである場合

```
1 class Pizza:
2     pass
3
4
5 class CheesePizza(Pizza):
6     pass
7
8
9 class BeconPizza(Pizza):
10    pass
11
12
13 class Drink:
14    pass
15
```

```
25 # 変数型がクラスで特定のクラスのみを許可したいの場合-----
26 # またはGenerics対処がクラスで特定のクラスのみ許可したい場合
27 from typing import Dict
28
29 pizza_menu1: Dict[str, Pizza] = {}
30 pizza_menu1["チーズピザ"] = CheesePizza() # Pizzaインスタンスしか渡せない。
31 pizza_menu1["チーズピザ"] = CheesePizza # Pizzaインスタンスしか渡せない。
32
33 pizza_menu2: Dict[str, type] = {} # isinstance(Pizza, type) == true
34 pizza_menu2["チーズピザ"] = CheesePizza # 全てのクラスがtypeのインスタンスのため、どんなクラスでも入れてしまう。
35 pizza_menu2["ベーコンピザ"] = BeconPizza
36 pizza_menu2["ドリンク"] = Drink # ピザのメニューなのにドリンクが入ってしまった！
37
38 def order_pizza(pizza_name: str) -> Pizza:
39     return pizza_menu2["ベーコンピザ"]()
```

Pizza_menuに入れるクラスをPizzaのサブクラスに制限したい場合どうする？

typing.Type

変数またはGenericsの対象がClassである場合

```
1 class Pizza:
2     pass
3
4
5 class CheesePizza(Pizza):
6     pass
7
8
9 class BeconPizza(Pizza):
10    pass
11
12
13 class Drink:
14    pass
15
```

```
43 # 変数型がクラスで特定のクラスのみを許可したいの場合-----
44 # またはGenerics対応がクラスで特定のクラスのみ許可したい場合
45 from typing import Type, Dict
46
47 # ケース 1
48 pizza_menu3: Dict[str, Type[Pizza]] = {}
49 pizza_menu3[""]
50 pizza_menu3["チーズピザ"] = CheesePizza # value型チェック : OK
51 pizza_menu3["ベーコンピザ"] = BeconPizza # value型チェック : OK
52 pizza_menu3["ドリンク"] = Drink # value型チェック : 警告、ドリンクはピザではない !
53
54 def order_pizza(pizza_name: str) -> Pizza:
55     return pizza_menu3["ベーコンピザ"]()
```

Static Type Checker

PyCharm v.s. Pylance(VSCode)



静的型検査ツールの違い

Pycharm v.s. Pylance (VS Code)

◆Pylance (VS Code) ではチェックされるが、PyCharmではチェックされない部分

```
8 def func(x: float) -> float: #警告: returnなしの可能性もある。PyCharmではチェックされない！
9     if x > 1:
10         return 1 #返回值型チェック: OK
11     elif x > 2:
12         return 1.0 #返回值型チェック: OK
13     elif x > 3:
14         return "a" #返回值型チェック: 警告
```

```
c: List[B1]
c = [B1()] #配列要素型チェック: OK
c = [C1()] #配列要素型チェック: OK
c = [A()] #配列要素型チェック: 警告
c = [B2()] #配列要素型チェック: 警告
c = [A(), B1()] #配列要素型チェック: 警告, PyCharmではB1要素が含まれていればOKになってしまう。
```

```
27 h: Callable[[A], float]
28 h = func1 #メソッドの形式チェック: OK
29 h = func2 #メソッドの形式チェック: 警告。引数のタイプが違う。サブクラスは含まない。
30 h = func3 #メソッドの形式チェック: 警告。リターンタイプが違う。
31 h = func4 #メソッドの形式チェック: 警告。引数のタイプが違う。
32 h = func5 #メソッドの形式チェック: 警告、引数数が違う。# Pycharmでは引数欠けている場合はOKとされてしまう。
```

※個人的にはPylance (VS Code)の方が静的型チェックのミスが少ないと感じた！

入力候補の違い

Pycharm v.s. VSCode

◆入力候補の違い（型定義していない場合）



◆ : よく使うキーワードも提示してくれる。

- + 型定義しなくても、一度出た名前が勝手に入力候補に出る
- 無関係な入力候補も出てしまう
- 入力候補に違う選択肢が出る可能性がある。

```
tmp.py x
1 def func(a):
2     a.
    if if expr
    ifn if expr is None
    ifnn if expr is not None
    main if __name__ == '__main__': expr
    not not expr
    par (expr)
    print print(expr)
    return return expr
    while while expr
    Press Enter to insert, Tab to replace Next Tip
```



◆ : 定義した型を根拠に候補を出す。

- + 入力候補の正確性が高い
- 型を定義していない場合何の候補も出ない


```
tmp.py •
typing_sample > tmp.py > func
1 def func(a):
2     a.
```

ありがとう ございました

[HTTP://WWW.MARGIESTRAVEL.COM/](http://www.margiestravel.com/)



 **VICTORIA LINDQVIST**

 +1 (589) 555-0199

 victoria@margiestravel.com

