python基础补充

*_dict_* attribute

```
# -*- coding: utf-8 -*-


class A(object):
    """
    Class A.
    """

    a = 0
    b = 1

    def __init__(self):
        self.a = 2
        self.b = 3

    def test(self):
        print ('a normal func.')

    @staticmethod
    def static_test(self):
        print ('a static func.')

    @classmethod
    def class_test(self):
        print ('a calss func.')


obj = A()
print(A.__dict__)
print(obj.__dict__)



{'__module__': '__main__', '__doc__': '\n    Class A.\n    ', 'a': 0, 'b': 1, '_
_init__': <function A.__init__ at 0x7f53ec1b0268>, 'test': <function A.test at 0
x7f53e4c74510>, 'static_test': <staticmethod object at 0x7f53ea87ff28>, 'class_t
```

```
est': <classmethod object at 0x7f53ea891048>, '__dict__': <attribute '__dict__'
of 'A' objects>, '__weakref__': <attribute '__weakref__' of 'A' objects>}
{'a': 2, 'b': 3}
```

我们可以看到A.**dict**可以获取到所有的属性以及方法而A的实例只能得到**init**方法下面定义的属性，在我的代码中，如何将一个python文件中的类以及类的属性和方法通过模块名直接引入到方法中，就是通过**dict**方法所起到的作用。

init_subclass

这是一个钩子方法，在注册子类的时候非常有用，可以用来在子类中添加默认的属性和方法。
例如：版本控制的适配器

```
class RepositoryType(Enum):
    HG = auto()
    GIT = auto()
    SVN = auto()
    PERFORCE = auto()
class Repository():
    _registry = {t: {} for t in RepositoryType}

    def __init_subclass__(cls, scm_type=None, name=None, **kwargs):
        super().__init_subclass__(**kwargs)
        if scm_type is not None:
            cls._registry[scm_type][name] = cls

class MainHgRepository(Repository, scm_type=RepositoryType.HG, name='main'):
    pass

class GenericGitRepository(Repository, scm_type=RepositoryType.GIT):
    pass
```

Generic Function in python with Singledispatch

> 想象一下，you can write different implementations of a function of the same name in the same scope,depending on thew types of arguments,Wouldn’t it be great?of course,it would be.There is a term for this.It is called “Generic Function”.python recently added support for generic function in python3.4 . They did this to the functools module by adding @singledipatch decorator.

what is Singledispatch

> A generic function is composed of multiple functions implementing the same operation for defferent types.which implementations should be used during a call operation for

> different types.Which implementation should be used during a call is determined by the dispatch algorithm.When the implementation is chosen based on the type of a single argument,this is known as single dispatch.

Singledispatch in Action

Let’s see singledispatch in action.There are few steps for writing a generic function with singledispatch.

- 从functools模块中引入singledispatch
- 通过@singledispatch装饰器定义一个默认或者回退函数，这是我们的通用性函数。
- 然后，通过类型注册额外的实现（Register additional implementations by passing the type in register() attribute of the generic function）所以你可以装饰你的实现通过 @function_name.register(type).You can also register lambdas and pre-existing functions.

现在，我们实现了一个通用性函数called fprint，可以打印一些东子基于类型，For list 它会答应 index 和值，for dict 它会答应key-value 对基于它的类型，默认情况下它会打印参数，

```
from functools import singledispatch
@singledispatch
def fprint(data):
    print(f'({type(data).__name__}) {data}')
```

注册list的实现

```
@fprint.register(list)
def _(data):
    formatted_header = f'{type(data).__name__} -> index : value'
    print(formatted_header)
    print('-' * len(formatted_header))
    for index, value in enumerate(data):
        print(f'{index} : ({type(value).__name__}) {value}')
```

以及其实现方式如下：

```
from functools import singledispatch
@singledispatch
def fprint(data):
    print(f'({type(data).__name__}) {data}')
@fprint.register(list)
@fprint.register(set)
@fprint.register(tuple)
def _(data):
    formatted_header = f'{type(data).__name__} -> index : value'
    print(formatted_header)
    print('-' * len(formatted_header))
    for index, value in enumerate(data):
        print(f'{index} : ({type(value).__name__}) {value}')
@fprint.register(dict)
def _(data):
```

```
    formatted_header = f'{type(data).__name__} -> key : value'
    print(formatted_header)
    print('-' * len(formatted_header))
    for key, value in data.items():
        print(f'({type(key).__name__}) {key}: ({type(value).__name__}) {value}')

# >>> fprint('hello')
# (str) hello

# >>> fprint(21)
# (int) 21

# >>> fprint(3.14159)
# (float) 3.14159

# >>> fprint([2, 3, 5, 7, 11])
# list -> index : value
# ---------------------
# 0 : (int) 2
# 1 : (int) 3
# 2 : (int) 5
# 3 : (int) 7
# 4 : (int) 11

# >>> fprint({2, 3, 5, 7, 11})
# set -> index : value
# --------------------
# 0 : (int) 2
# 1 : (int) 3
# 2 : (int) 5
# 3 : (int) 7
# 4 : (int) 11

# >>> fprint((13, 17, 23, 29, 31))
# tuple -> index : value
# ----------------------
# 0 : (int) 13
# 1 : (int) 17
# 2 : (int) 23
# 3 : (int) 29
# 4 : (int) 31

# >>> fprint({'name': 'John Doe', 'age': 32, 'location': 'New York'})
# dict -> key : value
# -------------------
# (str) name: (str) John Doe
# (str) age: (int) 32
# (str) location: (str) New York
```

4.typing —support for type hints

Note: The type module has beem include in the standard library on a provisional basis

```
def greeting(name:str) ->str:
    return 'Hello' + name
```

Type aliases:
通过将类型分配给别名来定义类型别名，在下面的例子中，Vector和List[float] 可以被看待为通用名词。

```
from typing import List
Vector = List[float]
def scale(scalar:float,vector:Vector) -> Vector:
    return [scalar * num for num in vector]
```

new_vector = scale(2.0,[1.0,-4.2,5.4])

Type aliases are useful for simplifying complex type signatures.for example:

```
from typing import Dict,Tuple,List
ConnectionOptions = Dict[str,str]
Address = Tuple[str,int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message:str,servers:List[Server]) ->None:
    ...
def broadcast_message(message:str,servers:List[Tuple[str,int],Dict[str,str]]]) -
> None:
```

NewType:
use the NewType() helper function to create distinct types:

```
from typing import NewType
UserId = NewType('UserId',int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of original type.This is useful in helping catch logical errors:

```
def get_user_name(user_id:UserId) -> str:
    ...
#typechecks
user_a = get_user_name(UserId(2224))
#does not typecheck;an int is not a UserId
user_b = get_user_name(-1)
```

You may still perform all int operations on a variable of type UserId,but the result will always be of type int.this lets you pass in a UserId whenever an int might be expected,but will prevent you from accidentally creating a UserId in a invalid way.

```
output = UserId(23413)+UserId(54341)
# ‘output’ is of type 'int',not 'UserId'
```

Note that these checks are enforced only by the static type checker. At runtime the statement Derived = NewType(‘Derived’,Base) will make Derived a function that immediately returns

whatever parameter you pass it.这也意味这 Derived(some_value) 不会创建一个新的类或者引入超出常规函数调用的任何开销。

callable

Generics

```
from typing import Mapping,Sequence
def notify_by_email(employee:Sequence[Employee],ovrrides:Mapping[str,str]) -> No
ne:...
```

User-defined generic types:
A user-defined class can be difined as a generic class.

```
from typing import TypeVar,Generic
from logging import Logger
T = TypeVar('T')
class LoggedVar(Generic[T]):
      def __init__(self,value:T,name:str,logger:Logger) -> None:
          self.name = name
          self.logger = logger
          self.value = value
      def set(self,new:T) -> None
           self.log('Set'+repr(self.value))
           self.value = new
       def get(self) ->T:
           self.log('Get：'+repr(self.value))
           return self.value
        def log(self,message:str) ->None
            self.logger.info()
```

Generic[T] as a base class defines that the class LoggedVar takes a single type parameter T.This also makes T valid as a type within the class body.
The Generic base class uses a metaclass that defines **getitem**() so that loggedVar[t] is valid as a type:

```
from typing import Iterable
def zero_all_vars(vars:Iterable[LoggedVar[int]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables,and type variables may be constrained:

```
from typing import TypeVar,Generic
...
T = TypeVar('T')
s = TypeVar('S',int,str)
class StrangePair(Generic[T,S]):
```

Each type variable argument to Generic must be distinct.this is thus invalid：

```
from typing import TypeVar,Generic
...
T = TypeVar('T')
class Pair(Generic[T,T]):##INVALID
...
```

you can use multiple inheritance with Generic:

```
from typing import TypeVar,Generic,Sized
T = TypeVar('T')
class LinkedList(Sized,Generic[T]):
      ...
```

when inheriting from generic classes,some type variables could be fixed:

```
from typing import TypeVar,Mapping
T = TypeVar('T')
class MyDict(Mapping[str,T]):
```

in this case Mydict has a single parameter,T.
User defined generic type aliases are also supported,Example:

```
from typing import TypeVar,Iterable,Tuple,Union
S = TypeVar('S')
Response = Union[Iterable[S],int]
#Return type here is same as Union[Iterable[str],int]
def response(query:str) -> Response[str]:
...
T = TypeVar('T',int,float,complex)
Vec = Iterable(Tuple[T,T])
def inproduct(v:Vec[T]) -> T:
     return sum(x*y for x,y in v)
```

The metaclass used by Generic is a subclass of abc.ABCMETA.A generic class can be an ABC by including abstract methods or properties,and generic classes can also have ABCs as base classes without a metaclass confilct.