PYTHON3 FUNCTION ANNOTATIONS

Michael Frank

syntaxjockey (at) gmail dot com

THE 5 SECOND OVERVIEW

- One of the new features of the python3 language is function annotations.
- Function annotations allow the developer to specify an arbitrary expression to be associated with a function's parameters or return type.

FOR EXAMPLE

```
>>> def func1(param1: int):
...     "Demonstrates a function with an annotated parameter”
...     pass

>>> def func2(param1: int) -> str:
...     "Demonstrates a function with an annotated parameter and return type”
...     pass

>>> def func3(param1: 2 + 2):
...     "Demonstrates that an annotation is just a python expression”
...     pass
```

FUNDAMENTALS 1

From the PEP:

"Function annotations, both for parameters and return values, are completely optional.”

"Function annotations are nothing more than a way of associating arbitrary Python expressions with various parts of a function at compile-time. By itself, Python does not attach any particular meaning or significance to annotations ... Python simply makes these expressions available."

FUNDAMENTALS 2

Annotations are made available through the __annotations__ attribute of the function

```
>>> def func1(param1: int):
...     "Demonstrates a function with an annotated parameter”
...     pass
>>> print(func1.__annotations__)
{'param1': <class 'int'>}
```

FUNDAMENTALS 3

All annotation expressions are evaluated when the function definition is executed, just like default values

```
>>> counter = 1
>>> def func2(param: counter + 1):
...     pass
>>> print(func2.__annotations__)
{'param': 2}
```

WHY CARE?

"[PEP-3107] makes no attempt to introduce any kind of standard semantics, even for the built-in types. This work will be left to third-party libraries."

So if function annotations are completely optional, and python3 doesn’t do anything with them when they are defined, then what is the purpose?

TYPE CHECKING

- Python is a dynamically-typed language
- However, sometimes you want to be explicit about the types of values a function receives and emits
- We can of course do all of this manually using `isinstance()` and `assert()`
- Python3 allows you to do so in a more intuitive manner with the help of external libraries
- One such library is Obiwan

USING OBIWAN

Install the obiwan type-checker at the beginning of your program

```
>>> from obiwan import *; install_obiwan_runtime_check()
```

Add type annotations to your methods

```
>>> def add2ints(i1: int, i2: int) -> int:
...     return i1 + i2
```

...PROFIT

```
>>> print(add2ints(1, 1))
2
>>> print(add2ints(1, 'hello'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in add2ints
  File "/usr/local/lib/python3.3/site-packages/obiwan/__init__.py", line 329, in
_runtime_checker
    duckable(arg, constraint, "%s(%s)" % (frame.f_code.co_name, key))
  File "/usr/local/lib/python3.3/site-packages/obiwan/__init__.py", line 251, in
duckable
    raise ObiwanError("%s is %s but should be %s" % (ctx, type(obj), template))
obiwan.ObiwanError: add2ints(i2) is <class 'str'> but should be <class 'int'>
```

MULTIMETHODS

- Multimethods (also known as multiple dispatch) are methods that have multiple versions, distinguished by type arguments
- Guido decribed a simple mechanism for multimethods in a blog post
- defines a @multimethod decorator that wraps functions

MULTIMETHOD EXAMPLE

```
from mm import multimethod

@multimethod(int, int)
def foo(a, b):
    ...code for two ints...

@multimethod(float, float):
def foo(a, b):
    ...code for two floats...

@multimethod(str, str):
def foo(a, b):
    ...code for two strings...
```

DRAWBACKS

- The previous mechanism works, but it has one significant drawback: the parameter type specification is decoupled from the function declaration
- Can we do better?

ANOTHER ATTEMPT

```
registry = {}

class MultiMethod(object):
    def __init__(self):
        self.typemap = {}
    def __call__(self, *args):
        types = tuple(arg.__class__ for arg in args)
        function = self.typemap.get(types)
        if function is None:
            raise TypeError("no match")
        return function(*args)
    def register(self, types, function):
        if types in self.typemap:
            raise TypeError("duplicate registration")
        self.typemap[types] = function
```

MULTIMETHOD CONTAINER

```
import inspect


def multimethod(function):
    name = function.__name__
    annotations = function.__annotations__
    mm = registry.get(name)
    if mm is None:
        mm = registry[name] = MultiMethod()
    signature = inspect.signature(function)
    types = tuple(annotations[param] for param in list(signature.parameters))
    mm.register(types, function)
    return mm
```

OVERLOADING FOO

```
@multimethod
def foo(a: int):
    return "just a lonely int"

@multimethod
def foo(a: int, b: str):
    return "an int and a string, what a perfect pair"

@multimethod
def foo(a: str, b: int):
    return "a string and an int, what a devilish combination"
```

MOMENT OF TRUTH

```
>>> print("foo(7) = {}".format(foo(7)))
foo(7) = just a lonely int

>>> print("foo(1,'a') = {}".format(foo(1,'a')))
foo(1,'a') = an int and a string, what a perfect pair

>>> print("foo('b',3) = {}".format(foo('b',3)))
foo('b',3) = a string and an int, what a devilish combination

>>> print("foo(3.14) = {}".format(foo(3.14)))
Traceback (most recent call last):
  File "./mm.py", line 47, in <module>
    print("foo(3.14) = {}".format(foo(3.14)))
  File "./mm.py", line 10, in __call__
    raise TypeError("no match")
TypeError: no match
```

LINKS

- Function annotations: http://legacy.python.org/dev/peps/pep-3107/
- Obiwan static type checker: https://pypi.python.org/pypi/obiwan
- Five minute multimethods in python: http://www.artima.com/weblogs/viewpost.jsp?thread=101605
- Function signatures (inspect module): http://legacy.python.org/dev/peps/pep-0362/
- Stackoverflow discussion of function annotations and their usage, including the inspiration for the multimethod example: http://stackoverflow.com/questions/3038033/what-are-good-uses-for-python3s-function-annotations