PyExt Documentation

Release 0.8

Ryan Gonzalez

July 09, 2015

Contents

Contents:

CHAPTER 1

Tutorial

1.1 Function overloading

PyExt function overloading is simple:

```
@overload.argc(1)
def x(a): print 'Function 1 called'

@overload.argc(2)
def x(a, b): print 'Function 2 called'

x(1)
x(1, 2)
```

Easy, no? Now, if we wanted to overload by types also, we could do this:

```
@overload.args(int)
def x(a): print 'Function 1 called'

@overload.args(str)
def x(a): print 'Function 2 called'

x(1)
x('s')
```

If you're in Python 3, you can also use function annotations by passing `None` as the parameter:

```
@overload.args(None)
def x(a:int): print 'Function 1 called'

@overload.args(None)
def x(a:str): print 'Function 2 called'

x(1)
x('s')
```

If you are overloading a class method, you need to pass `is_cls` as `True`:

```
class x(object):
    @overload.args(str, is_cls=True)
    def f(self, s): print 'Got string'
    @overload.args(int, is_cls=True)
    def f(self, i): print 'Got int'
```

```
obj = x()
x.f('s')
x.f(2)
```

1.2 Runtime Modules

Runtime modules let you create a full module object at runtime. Here's an example:

```
mymodule = RuntimeModule.from_objects('module_name', 'module_docstring', a=1, b=2)
import mymodule # Module object is added to sys.path
print mymodule.a, mymodule.b
```

We can also create our module object from a string:

```
mystr = '''
a = 1
b = 2
def do_nothing(x): return 'Nothing'
'''
RuntimeModule.from_string('module_name', 'module_docstring', mystr)
import mymodule
print mymodule.a, mymodule.b, mymodule.do_nothing(1)
```

1.3 Switch statement

Switch statements are just as easy as everything else:

```
with switch(3):
    if case(1): print 'Huh?'
    if case(2): print 'What the...'
    if case(3): print "That's better!"
    if case.default(): print 'Ummm...'
```

This is equilavent to the following C code:

```
switch(3)
{
case 1:
  puts("Huh?"); break;
case 2:
  puts("What the..."); break;
case 3:
  puts("That's better!"); break;
default:
  puts("Ummm...")
}
```

The equivalent of `break` is `case.quit()`. There is an implicit quit at the end of every case. If you want the C-style fallthrough, pass `cstyle=True` to the call to `switch`.

For chaining `case` statements, pass multiple arguments to case. For example, this C:

```
switch(myvar)
{
case 1:
```

```
case 3:
case 5:
case 7:
case 9:
  puts("An odd number"); break;
case 2:
case 4:
case 6:
case 8:
  puts("An even number"); break;
default:
  puts("The number is either greater that 9 or less than 1");
}
```

is equilavent to this Python code using PyExt:

```
with switch(myvar):
    if case(1,3,5,7,9): print 'An odd number'
    if case(2,4,6,8): print 'An even number'
    if case.default(): print 'The number is either greater that 9 or less than 1'
```

1.4 Tail recursion removal

Have you ever had a function that went way over the recursion limit? PyExt has a feature that eliminates that problem:

```
@tail_recurse()
def add(a, b):
    if a == 0: return b
    return add(a-1, b+1)

add(1000000, 1) # Doesn't max the recursion limit!
```

1.5 Function annotations

PyExt lets you use Python 3's function annotations...on Python 2! Here is an example:

```
@fannotate('ret', a='a', b=1,)
def x(a, b):
   return 0
```

This is equilavent to:

```
def x(a:'a', b:1) -> 'ret':
   return 0
```

1.6 Safe unpacking

Say you have a string whose value is `'a:b'`. Now, say you want to split this string at the colon. You'll probably do this:

```
a, b = my_string.split(':')
```

But what if `my_string` doesn't have a colon? You'll have to do this:

```
a, b = my_string.split(':') if ':' in my_string else (my_string, None)
```

Python 3 lets you simply do this:

```
a, *b = my_string.split(':')
```

Also, with string partitioning, you can do this:

```
a, _, b = my_string.partition(':')
```

But say you're not working on a string. Say you're using a tuple:

```
a, b = my_tuple
```

If my_tuple isn't big enough or is too big, it'll throw an error. As stated above, Python 3 fixes this. But what if you're using Python 2? PyExt comes with a nifty function called `safe_unpack` that lets you do this:

```
a, b = safe_unpack(my_tuple, 2)
```

The first parameter is the sequence to unpack, while the second is the expected length. If the sequence is too large, the excess values are ignored. If it's too small, `None` is substituted in for the extra values.

You can also specify a value other than `None` to fill in the extra spaces:

```
a, b = safe_unpack(my_tuple, 2, fill='')
```

1.7 Expression assignment

Languages such as C and C++ allow you to use an assignment as an expression. For people who don't know what that means, here's an example, in C:

```
if (my_var = my_func())
```

This is equilavent to the following Python code:

```
my_var = my_func()
if my_var:
```

PyExt lets you do it the easy way:

```
if assign('my_var', my_func()):
```

1.8 Swap if condition

Code like this is more than often encountered:

```
if my_variable == some_value:
    my_variable = some_other_value
```

Using PyExt, that code gets shortened to:

```
compare_and_swap('my_variable', some_value, some_other_value)
```

CHAPTER 2

PyExt API Docs

class pyext.**overload**

Simple function overloading in Python.

classmethod argc(*argc=None*)

Overloads a function based on the specified argument count.

Parameters **argc** – The argument count. Defaults to None. If None is given, automatically compute the argument count from the given function.

Note: Keyword argument counts are NOT checked! In addition, when the argument count is automatically calculated, the keyword argument count is also ignored!

Example:

```
@overload.argc()
def func(a):
    print 'Function 1 called'

@overload.argc()
def func(a, b):
    print 'Function 2 called'

func(1) # Calls first function
func(1, 2) # Calls second function
func() # Raises error
```

classmethod args(**argtypes*, ***kw*)

Overload a function based on the specified argument types.

Parameters

- **argtypes** – The argument types. If None is given, get the argument types from the function annotations(Python 3 only)
- **kw** – Can only contain 1 argument, *is_cls*. If True, the function is assumed to be part of a class.

Example:

```
@overload.args(str)
def func(s):
    print 'Got string'

@overload.args(int, str)
```

```
        def func(i, s):
            print 'Got int and string'

        @overload.args()
        def func(i:int): # A function annotation example
            print 'Got int'

        func('s')
        func(1)
        func(1, 's')
        func(True) # Raises error
```

class pyext.**RuntimeModule**
Create a module object at runtime and insert it into sys.path. If called, same as `from_objects()`.

pyext.**switch**(*value*, *cstyle=False*)
A Python switch statement implementation that is used with a `with` statement.

Parameters

- **value** – The value to "switch".
- **cstyle** – If `True`, then cases will automatically fall through to the next one until `case.quit()` is encountered.

`with` statement example:

```
    with switch('x'):
        if case(1): print 'Huh?'
        if case('x'): print 'It works!!!'
```

Warning: If you modify a variable named "case" in the same scope that you use the `with` statement version, you will get an UnboundLocalError. The soluction is to use `with switch('x') as case:` instead of `with switch('x'):`.

pyext.**tail_recurse**(*spec=None*)
Remove tail recursion from a function.

Parameters **spec** – A function that, when given the arguments, returns a bool indicating whether or not to exit. If `None`, tail recursion is always called unless the function returns a value.

Note: This function has a slight overhead that is noticable when using timeit. Only use it if the function has a possibility of going over the recursion limit.

Warning: This function will BREAK any code that either uses any recursion other than tail recursion or calls itself multiple times. For example, `def x(): return x()+1` will fail.

Example:

```
    @tail_recurse()
    def add(a, b):
        if a == 0: return b
        return add(a-1, b+1)

    add(10000000, 1) # Doesn't max the recursion limit.
```

pyext.**copyfunc**(*f*)
Copies a funcion.

Parameters **f** – The function to copy.

Returns The copied function.

Deprecated since version 0.4: Use *modify_function()* instead.

pyext.**set_docstring**(*doc*)
A simple decorator to set docstrings.

Parameters **doc** – The docstring to tie to the function.

Example:

```
@set_docstring('This is a docstring')
def myfunc(x):
    pass
```

pyext.**annotate**(**args*, ***kwargs*)
Set function annotations using decorators.

Parameters

- **args** – This is a list of annotations for the function, in the order of the function's parameters. For example, `annotate('Annotation 1', 'Annotation 2')` will set the annotations of parameter 1 of the function to `Annotation 1`.
- **kwargs** – This is a mapping of argument names to annotations. Note that these are applied *after* the argument list, so any args set that way will be overriden by this mapping. If there is a key named *ret*, that will be the annotation for the function's return value.

Deprecated since version 0.5: Use *fannotate()* instead.

pyext.**safe_unpack**(*seq*, *ln*, *fill=None*)
Safely unpack a sequence to length *ln*, without raising ValueError. Based on Lua's method of unpacking. Empty values will be filled in with *fill*, while any extra values will be cut off.

Parameters

- **seq** – The sequence to unpack.
- **ln** – The expected length of the sequence.
- **fill** – The value to substitute if the sequence is too small. Defaults to `None`.

Example:

```
s = 'a:b'
a, b = safe_unpack(s.split(':'), 2)
# a = 'a'
# b = 'b'
s = 'a'
a, b = safe_unpack(s.split(':'), 2)
# a = 'a'
# b = None
```

pyext.**modify_function**(*f*, *globals={}*, *name=None*, *code=None*, *defaults=None*, *closure=None*)
Creates a copy of a function, changing its attributes.

Parameters

- **globals** – Will be added to the function's globals.
- **name** – The new function name. Set to `None` to use the function's original name.

- **code** – The new function code object. Set to `None` to use the function's original code object.
- **defaults** – The new function defaults. Set to `None` to use the function's original defaults.
- **closure** – The new function closure. Set to `None` to use the function's original closure.

Warning: This function can be potentially dangerous.

pyext.**assign**(*varname*, *value*)

Assign *value* to *varname* and return it. If *varname* is an attribute and the instance name it belongs to is not defined, a NameError is raised. This can be used to emulate assignment as an expression. For example, this:

```
if assign('x', 7): ...
```

is equilavent to this C code:

```
if (x = 7) ...
```

Warning: When assigning an attribute, the instance it belongs to MUST be declared as global prior to the assignment. Otherwise, the assignment will not work.

pyext.**fannotate**(**args*, ***kwargs*)

Set function annotations using decorators.

Parameters

- ***args** – The first positional argument is used for the function's return value; all others are discarded.
- ****kwargs** – This is a mapping of argument names to annotations.

Example:

```
@fannotate('This for the return value', a='Parameter a', b='Parameter b')
def x(a, b):
    pass
```

pyext.**compare_and_swap**(*var*, *compare*, *new*)

If *var* is equal to *compare*, set it to *new*.

pyext.**is_main**(*frame=1*)

Return if the caller is main. Equilavent to `__name__ == '__main__'`.

pyext.**call_if_main**(*f*, **args*)

Call the *f* with *args* if the caller's module is main.

pyext.**run_main**(*f*, **args*)

Call *f* with the *args* and terminate the program with its return code if the caller's module is main.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

A

C

F

I

M

O

P

R

S

T