Add a variable to a class instance at runtime

Demonstrate how to dynamically add variables to an object (a class instance) at runtime.

This is useful when the methods/variables of an instance are based on a data file that isn't available until runtime. Hal Fulton gives an example of creating an OO CSV parser at An Exercise in Metaprogramming with Ruby (http://www.devsource.com/article2/0,1759,1928562,00.asp). This is referred to as "monkeypatching" by Pythonistas and some others.


(/wiki/Category:Solutions_by_Programming_Task)

Add a variable to a class instance at runtime
You are encouraged to solve this task (/wiki/Rosetta_Code:Solve_a_Task) according to the task description, using any language you may know.

Contents

```
PS> $x | Get-Member

   TypeName: System.Int32

Name        MemberType   Definition
----        ----------   ----------
CompareTo   Method       int CompareTo(System.Object value), ...
Equals      Method       bool Equals(System.Object obj), bool...
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
GetTypeCode Method       System.TypeCode GetTypeCode()
ToString    Method       string ToString(), string ToString(s...
Title       NoteProperty System.String Title=The answer to th...
```

While trying to access the same property in another instance will fail:

```
PS> $y = 42
PS> $y.Title
```

(which simply causes no output).

Python (/wiki/Category:Python)

```
class empty(object):
    pass
e = empty()
```

If the variable (attribute) name is known at "compile" time (hard-coded):

```
   e.foo = 1
```

If the variable name is determined at runtime:

```
   setattr(e, name, value)
```

Note: Somewhat counter-intuitively one cannot simply use *e = object(); e.foo = 1* because the Python base *object* (the ultimate ancestor to all new-style classes) will raise attribute exceptions. However, any normal derivatives of *object* can be "monkey patched" at will.

Because functions are first class objects in Python one can not only add variables to instances. One can add or replace functionality to an instance. Doing so is tricky if one wishes to refer back to other instance attributes since there's no "magic" binding back to "self." One trick is to dynamically define the function to be added, nested within the function that applies the patch like so:

```
class empty(object):
    def __init__(this):
        this.foo = "whatever"

def patch_empty(obj):
```

```
    def fn(self=obj):
        print self.foo
    obj.print_output = fn

e = empty()
patch_empty(e)
e.print_output()
# >>> whatever
```

Note: The name *self* is not special; it's merely the pervasive Python convention. In this example I've deliberately used *this* in the class definition to underscore this fact. The nested definition could use any name for the "self" object. Because it's nested the value of the object is evaluated at the time that the patch_empty() function is run and thus the function being patched in has a valid reference to the object into which it is being inserted. Other arguments could be passed as necessary. Such techniques are not recommended; however they are possible.

REBOL (/wiki/Category:REBOL)

```
rebol [
        Title: "Add Variables to Class at Runtime"
        Author: oofoe
        Date: 2009-12-04
        URL: http://rosettacode.org/wiki/Adding_variables_to_a_class_instance_at_r
untime
]

; As I understand it, a REBOL object can only ever have whatever
; properties it was born with. However, this is somewhat offset by the
; fact that every instance can serve as a prototype for a new object.
```