See What the Builtins Are (Python recipe) by Eric Snow

ActiveState Code (http://code.activestate.com/recipes/577888/)

▲
1
▼

The built-ins are all those functions, constants, and types that are there when you need them most, like `list`, `any`, and `None`. This recipe is a simple way of looking at the built-ins. Check below for a thorough explanation.

Python, 48 lines

```
"""show_builtins module"""

import types
import collections
try:
    import builtins
except ImportError:
    builtins = __builtin__

ODict = collections.OrderedDict


def show_builtins():

    keys = ("Warnings", "Exceptions", "Types", "Functions", "Others")
    objs = dict(zip(keys, (ODict() for i in keys)))

    for name in dir(builtins):
        if name in ("__doc__", "__name__", "__package__"):
            continue

        obj = getattr(builtins, name)
        _repr = "<{} object>".format(type(obj).__name__)
        if isinstance(obj, type):
            if issubclass(obj, Warning):
                objs["Warnings"][name] = obj
            elif issubclass(obj, Exception):
                objs["Exceptions"][name] = obj
            else:
                objs["Types"][name] = obj
        elif callable(obj):
            if isinstance(obj, types.FunctionType):
                objs["Functions"][name] = obj
            else:
                objs["Functions"][name] = _repr
        else:
            objs["Others"][name] = _repr

    for key in keys:
        section_name = " Builtin {}:".format(key)
        print()
        print(section_name)
        print("-"*(len(section_name)+1))
        for name, obj in objs[key].items():
            print("{:<21} {}".format(name, obj))

if __name__ == "__main__":
    show_builtins()
```

The built-ins are the functions, constants, types, and exception classes that we take for granted as always being there for us. Here's a breakdown:

2.7 ('__builtin__' module): functions, constants, types , and exception classes;

3.2 ('builtins' module): functions, constants, types , and exception classes.

Code Execution and Name Lookup in Python

During code execution in Python, names are first looked for in the local namespace of the execution frame. If not found there, the global namespace bound to that frame is checked. Finally, if still not found, the built-in namespace that's bound to the frame is checked.

Not only does this apply as expected to function calls, but it also includes modules at import time, classes at definition time, exec calls, and the interactive interpreter. You can read all about the Python execution model at this page in the Python documentation.

Regardless of the code block, the globals are exposed there by the built-in globals() function. The locals are exposed there by the built-in locals() and vars() functions. The locals should be considered a read-only namespace, as there is no language guarantee that changes you make to it directly will actually be applied.

For some code blocks, like function and class bodies, the globals and locals are distinct; and the locals are inaccessible from outside that code block. For other code, like modules, they are the same thing (and thus equally accessible). With exec(), you can pass both in, thus controlling the distinction.

More on Name Lookup

The context for the built-ins is the *implicit* name lookups for unqualified names (we introduced lookups earlier). As noted, the built-ins are the last namespace to be checked during name lookup.

They are preceeded in that lookup chain by the globals, effectively always the namespace of the module in which the code was *defined* (rather than where the code is *running*). For modules the globals are the module object's own namespace (e.g. <module>.__dict__).

The globals passed to exec() are the exception, as they are any mapping you feel like passing there.

As we've already implied, if a name is unqualified (not *syntactically* tied to another object), then it gets looked up through the lookup chain of the execution frame. Python also provides syntax for explicit lookups on other namespaces found in the execution frame (usually bound to names there).

Principal among these are dotted lookups (`<OBJ>.<ATTR>`) and indexed access (`<OBJ>[<KEY>]`). These lookup mechanisms are separate from the lookups on the execution frame. In contrast they are looking up names, respectively, on the object's own namespace (usually `__dict__`) and on the mapping represented by the object.

Dotted lookup is handled by the interpreter through the `__getattribute__()` method (and siblings) of objects. Indexed lookup is likewise handled through the `__getitem__()` method (and siblings). These are explained in detail on the data model page of the Python language reference. There you'll also find some special caveats about special method names.

The Built-ins

The built-in names are your good friends. They've seen you through thick and thin. Seriously, take a look at the lists of above. Maybe you've wondered where they come from (or not). Maybe you just assumed that their just compiled in with some special efficient handling. Not so much.

Almost universally we take for granted that the built-ins are mostly just everyday functions like you or

I might write (albeit optimized and in C <wink>). Do we think about where None comes from every time we check for it? No? Okay.

The built-ins are the the last resort for unqualified name lookup in an execution frame. They are simply a namespace like any other, basically a mapping, perhaps a dictionary. When an execution frame runs, it tries to pull `__builtins__` from the globals namespace it was sent. If it doesn't find it there, the interpreter will automatically use the `__dict__` of the `__builtin__` module (in 2.x; `builtins` in 3.x).

So there you have it. The built-ins revealed. If you want to, feel free to import the appropriate module for your Python version and take a look inside. You can add anything you like there and it will suddenly be available in all your code! Do that in your sitecustomize.py and you've suddenly made your code less portable^W^Wmuch cleaner <wink>.

If you've never liked the way Python imports work, just replace `__builtin__.__import__` with your own function that does what you want.

If you really feel daring replace other stuff with your own versions.

Stuck on Python 2.4 but wish you had `any()`? Write your own and add it to `__builtin__`. Suddenly your code is more forward compatible.

The sky's the limit.

Just be aware that if you replace something like None, things might stop working. Alternately, the underlying Python implementation *may* use the old value directly instead of using the one you just set. That's not unlike how CPython's default __import__ directly uses the dict bound to sys.modules, rather than pulling it from sys.modules. If you replace sys.modules with your own mapping, it gets ignored.

One Trick

Using built-ins directly during execution is less efficient than using locals or globals. This is because the built-ins are at the end of the lookup chain, so the lookups on the locals and globals are a waste. One solution is to bind a local name to the built-in you want to use:

```
names = [...] # some list of names
_len = len
for name in names:
    print("Name: %s" % name)
    if _len(name) > 15:
        print(" *** Wow!  That's a long name! ***")
```

You can do the same thing with other lookups, e.g. global, dotted, and indexed names, to the same effect.

A variation of this trick is used sometimes with function default arguments (a.k.a. the "default argument hack"). The default argument values are evaluated as definition time and *not* each time the function is called. Loading the default argument values at call-time is much faster than doing the global/built-in lookup at each call.

One Warning

When code executes (at least in CPython), it runs in an execution frame. The globals and built-ins that the code should use are bound in that frame.

When a module is executed during import (at least in CPython), the built-ins are bound in the module's namespace as __builtins__. This is because the module's `__dict__` is used as the globals namespace and the built-ins live there during execution. Thus, this binding leaks out in your module code and you can access `__builtins__` directly there as an unqualified name.

The problem is that it may not always happen and it may not be reliable. The __builtin__ module documentation spells this out for you:

```
As an implementation detail, most modules have the name __builtins__
made available as part of their globals. The value of __builtins__ is
normally either this module or the value of this modules's __dict__
attribute. Since this is an implementation detail, it may not be used
by alternate implementations of Python.
```

So, just to reiterate, you should **not** use `__builtins__`. Instead import and use `__builtin__` (2.x) or `builtin` (3.x).

Tags: builtins

3 comments

zetah 4 years, 8 months ago

Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24) [GCC 4.5.2] on linux2

ImportError: No module named builtins

Eric Snow (author) 4 years, 8 months ago

Ah yes, "builtins" was added in Python 3. In 2.7 you can use __builtins__ as an alternative. I've updated the recipe to address this. Thanks for pointing this out.

Eric Snow (author) 4 years, 8 months ago

Caught a mistake (used __builtins__ instead of __builtin__ for 2.7).