10.8. `functools` — Higher order functions and operations on callable objects¶

New in version 2.5.

The `functools` module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The `functools` module defines the following functions:

`functools.reduce`(*function*, *iterable*[, *initializer*])¶

This is the same function as `reduce()`. It is made available in this module to allow writing code more forward-compatible with Python 3.

New in version 2.6.

`functools.partial`(*func*[, **args*][, ***keywords*])¶

Return a new `partial` object which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, *args, **keywords):
   def newfunc(*fargs, **fkeywords):
       newkeywords = keywords.copy()
       newkeywords.update(fkeywords)
       return func(*(args + fargs), **newkeywords)
   newfunc.func = func
   newfunc.args = args
   newfunc.keywords = keywords
   return newfunc
```

The `partial()` is used for partial function application which “freezes” some portion of a function’s arguments and/or keywords resulting in a new object with a simplified signature. For example, `partial()` can be used to create a callable that behaves like the `int()` function where the *base* argument defaults to two:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

`functools.update_wrapper`(*wrapper*, *wrapped*[, *assigned*][, *updated*])¶

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants *WRAPPER_ASSIGNMENTS* (which assigns to the wrapper function’s *__name__*, *__module__* and *__doc__*, the documentation string) and *WRAPPER_UPDATES* (which updates the wrapper function’s *__dict__*, i.e. the instance dictionary).

The main intended use for this function is in *decorator* functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

`functools.wraps`(*wrapped*[, *assigned*][, *updated*])¶

This is a convenience function for invoking `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)` as a function decorator when defining a wrapper function. For example:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
```

```
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been `'wrapper'`, and the docstring of the original `example()` would have been lost.

10.8.1. `partial` Objects¶

`partial` objects are callable objects created by `partial()`. They have three read-only attributes:

`partial.func`¶

A callable object or function. Calls to the `partial` object will be forwarded to `func` with new arguments and keywords.

`partial.args`¶

The leftmost positional arguments that will be prepended to the positional arguments provided to a `partial` object call.

`partial.keywords`¶

The keyword arguments that will be supplied when the `partial` object is called.

`partial` objects are like `function` objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the `__name__` and `__doc__` attributes are not created automatically. Also, `partial` objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.