:mod:`functools` --- Higher-order functions and operations on callable objects

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 4)

Unknown directive type "module".

```
.. module:: functools
   :synopsis: Higher-order functions and operations on callable objects.
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 7)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Peter Harris <scav@blueyonder.co.uk>
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 8)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Raymond Hettinger <python@rcn.com>
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 9)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Nick Coghlan <ncoghlan@gmail.com>
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 10)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Łukasz Langa <lukasz@langa.pl>
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 11)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Pablo Galindo <pablogsal@gmail.com>
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 12)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Peter Harris <scav@blueyonder.co.uk>
```

Source code: :source:`Lib/functools.py`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 14); *backlink*

Unknown interpreted text role "source".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 16)

Unknown directive type "testsetup".

```
.. testsetup:: default

   import functools
   from functools import *
```

The :mod:`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.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 23); *backlink*

Unknown interpreted text role "mod".

The :mod:`functools` module defines the following functions:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 27); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 29)

Unknown directive type "decorator".

```
.. decorator:: cache(user_function)

   Simple lightweight unbounded function cache.  Sometimes called
   `"memoize" <https://en.wikipedia.org/wiki/Memoization>`_.

   Returns the same as ``lru_cache(maxsize=None)``, creating a thin
   wrapper around a dictionary lookup for the function arguments.  Because it
   never needs to evict old values, this is smaller and faster than
   :func:`lru_cache()` with a size limit.

   For example::

        @cache
        def factorial(n):
            return n * factorial(n-1) if n else 1

        >>> factorial(10)      # no previously cached result, makes 11 recursive calls
        3628800
        >>> factorial(5)       # just looks up cached value result
        120
        >>> factorial(12)      # makes two new recursive calls, the other 10 are cached
        479001600

   .. versionadded:: 3.9
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 55)

Unknown directive type "decorator".

```
.. decorator:: cached_property(func)

   Transform a method of a class into a property whose value is computed once
   and then cached as a normal attribute for the life of the instance. Similar
   to :func:`property`, with the addition of caching. Useful for expensive
   computed properties of instances that are otherwise effectively immutable.

   Example::

       class DataSet:

           def __init__(self, sequence_of_numbers):
               self._data = tuple(sequence_of_numbers)
```

```
            @cached_property
            def stdev(self):
                return statistics.stdev(self._data)

    The mechanics of :func:`cached_property` are somewhat different from
    :func:`property`.  A regular property blocks attribute writes unless a
    setter is defined. In contrast, a *cached_property* allows writes.

    The *cached_property* decorator only runs on lookups and only when an
    attribute of the same name doesn't exist.  When it does run, the
    *cached_property* writes to the attribute with the same name. Subsequent
    attribute reads and writes take precedence over the *cached_property*
    method and it works like a normal attribute.

    The cached value can be cleared by deleting the attribute.  This
    allows the *cached_property* method to run again.

    Note, this decorator interferes with the operation of :pep:`412`
    key-sharing dictionaries.  This means that instance dictionaries
    can take more space than usual.

    Also, this decorator requires that the ``__dict__`` attribute on each instance
    be a mutable mapping. This means it will not work with some types, such as
    metaclasses (since the ``__dict__`` attributes on type instances are
    read-only proxies for the class namespace), and those that specify
    ``__slots__`` without including ``__dict__`` as one of the defined slots
    (as such classes don't provide a ``__dict__`` attribute at all).

    If a mutable mapping is not available or if space-efficient key sharing
    is desired, an effect similar to :func:`cached_property` can be achieved
    by a stacking :func:`property` on top of :func:`cache`::

        class DataSet:
            def __init__(self, sequence_of_numbers):
                self._data = sequence_of_numbers

            @property
            @cache
            def stdev(self):
                return statistics.stdev(self._data)

    .. versionadded:: 3.8
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst, line 113)

Unknown directive type "function".

```
.. function:: cmp_to_key(func)

    Transform an old-style comparison function to a :term:`key function`.  Used
    with tools that accept key functions (such as :func:`sorted`, :func:`min`,
    :func:`max`, :func:`heapq.nlargest`, :func:`heapq.nsmallest`,
    :func:`itertools.groupby`).  This function is primarily used as a transition
    tool for programs being converted from Python 2 which supported the use of
    comparison functions.

    A comparison function is any callable that accept two arguments, compares them,
    and returns a negative number for less-than, zero for equality, or a positive
    number for greater-than.  A key function is a callable that accepts one
    argument and returns another value to be used as the sort key.

    Example::

        sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order

    For sorting examples and a brief sorting tutorial, see :ref:`sortinghowto`.

    .. versionadded:: 3.2
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst, line 136)

Unknown directive type "decorator".

```
.. decorator:: lru_cache(user_function)
               lru_cache(maxsize=128, typed=False)
```

```
Decorator to wrap a function with a memoizing callable that saves up to the
*maxsize* most recent calls.  It can save time when an expensive or I/O bound
function is periodically called with the same arguments.

Since a dictionary is used to cache results, the positional and keyword
arguments to the function must be hashable.

Distinct argument patterns may be considered to be distinct calls with
separate cache entries.  For example, `f(a=1, b=2)` and `f(b=2, a=1)`
differ in their keyword argument order and may have two separate cache
entries.

If *user_function* is specified, it must be a callable. This allows the
*lru_cache* decorator to be applied directly to a user function, leaving
the *maxsize* at its default value of 128::

    @lru_cache
    def count_vowels(sentence):
        return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')

If *maxsize* is set to ``None``, the LRU feature is disabled and the cache can
grow without bound.

If *typed* is set to true, function arguments of different types will be
cached separately.  If *typed* is false, the implementation will usually
regard them as equivalent calls and only cache a single result. (Some
types such as *str* and *int* may be cached separately even when *typed*
is false.)

Note, type specificity applies only to the function's immediate arguments
rather than their contents.  The scalar arguments, ``Decimal(42)`` and
``Fraction(42)`` are be treated as distinct calls with distinct results.
In contrast, the tuple arguments ``('answer', Decimal(42))`` and
``('answer', Fraction(42))`` are treated as equivalent.

The wrapped function is instrumented with a :func:`cache_parameters`
function that returns a new :class:`dict` showing the values for *maxsize*
and *typed*.  This is for information purposes only.  Mutating the values
has no effect.

To help measure the effectiveness of the cache and tune the *maxsize*
parameter, the wrapped function is instrumented with a :func:`cache_info`
function that returns a :term:`named tuple` showing *hits*, *misses*,
*maxsize* and *currsize*.

The decorator also provides a :func:`cache_clear` function for clearing or
invalidating the cache.

The original underlying function is accessible through the
:attr:`__wrapped__` attribute.  This is useful for introspection, for
bypassing the cache, or for rewrapping the function with a different cache.

The cache keeps references to the arguments and return values until they age
out of the cache or until the cache is cleared.

An `LRU (least recently used) cache
<https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)>`_
works best when the most recent calls are the best predictors of upcoming
calls (for example, the most popular articles on a news server tend to
change each day).  The cache's size limit assures that the cache does not
grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse
previously computed values.  Accordingly, it doesn't make sense to cache
functions with side-effects, functions that need to create distinct mutable
objects on each call, or impure functions such as time() or random().

Example of an LRU cache for static web content::

     @lru_cache(maxsize=32)
     def get_pep(num):
         'Retrieve text of a Python Enhancement Proposal'
         resource = 'https://peps.python.org/pep-%04d/' % num
         try:
             with urllib.request.urlopen(resource) as s:
                 return s.read()
         except urllib.error.HTTPError:
             return 'Not Found'

     >>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
     ...     pep = get_pep(n)
     ...     print(n, len(pep))
```

```
    >>> get_pep.cache_info()
    CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)

Example of efficiently computing
`Fibonacci numbers <https://en.wikipedia.org/wiki/Fibonacci_number>`_
using a cache to implement a
`dynamic programming <https://en.wikipedia.org/wiki/Dynamic_programming>`_
technique::

    @lru_cache(maxsize=None)
    def fib(n):
        if n < 2:
            return n
        return fib(n-1) + fib(n-2)

    >>> [fib(n) for n in range(16)]
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

    >>> fib.cache_info()
    CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)

.. versionadded:: 3.2

.. versionchanged:: 3.3
   Added the *typed* option.

.. versionchanged:: 3.8
   Added the *user_function* option.

.. versionadded:: 3.9
   Added the function :func:`cache_parameters`
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 254)

Unknown directive type "decorator".

```
.. decorator:: total_ordering

   Given a class defining one or more rich comparison ordering methods, this
   class decorator supplies the rest.  This simplifies the effort involved
   in specifying all of the possible rich comparison operations:

   The class must define one of :meth:`__lt__`, :meth:`__le__`,
   :meth:`__gt__`, or :meth:`__ge__`.
   In addition, the class should supply an :meth:`__eq__` method.

   For example::

       @total_ordering
       class Student:
           def _is_valid_operand(self, other):
               return (hasattr(other, "lastname") and
                       hasattr(other, "firstname"))
           def __eq__(self, other):
               if not self._is_valid_operand(other):
                   return NotImplemented
               return ((self.lastname.lower(), self.firstname.lower()) ==
                       (other.lastname.lower(), other.firstname.lower()))
           def __lt__(self, other):
               if not self._is_valid_operand(other):
                   return NotImplemented
               return ((self.lastname.lower(), self.firstname.lower()) <
                       (other.lastname.lower(), other.firstname.lower()))

   .. note::

      While this decorator makes it easy to create well behaved totally
      ordered types, it *does* come at the cost of slower execution and
      more complex stack traces for the derived comparison methods. If
      performance benchmarking indicates this is a bottleneck for a given
      application, implementing all six rich comparison methods instead is
      likely to provide an easy speed boost.

   .. note::

      This decorator makes no attempt to override methods that have been
      declared in the class *or its superclasses*. Meaning that if a
      superclass defines a comparison operator, *total_ordering* will not
      implement it again, even if the original method is abstract.

   .. versionadded:: 3.2
```

```
.. versionchanged:: 3.4
   Returning NotImplemented from the underlying comparison function for
   unrecognised types is now supported.
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 304)

Unknown directive type "function".

```
.. function:: partial(func, /, *args, **keywords)

   Return a new :ref:`partial object<partial-objects>` 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, **fkeywords}
              return func(*args, *fargs, **newkeywords)
          newfunc.func = func
          newfunc.args = args
          newfunc.keywords = keywords
          return newfunc

   The :func:`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, :func:`partial` can be used to create
   a callable that behaves like the :func:`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
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 335)

Invalid class attribute value for "class" directive: "partialmethod(func, /, *args, **keywords)".

```
.. class:: partialmethod(func, /, *args, **keywords)

   Return a new :class:`partialmethod` descriptor which behaves
   like :class:`partial` except that it is designed to be used as a method
   definition rather than being directly callable.

   *func* must be a :term:`descriptor` or a callable (objects which are both,
   like normal functions, are handled as descriptors).

   When *func* is a descriptor (such as a normal Python function,
   :func:`classmethod`, :func:`staticmethod`, :func:`abstractmethod` or
   another instance of :class:`partialmethod`), calls to ``__get__`` are
   delegated to the underlying descriptor, and an appropriate
   :ref:`partial object<partial-objects>` returned as the result.

   When *func* is a non-descriptor callable, an appropriate bound method is
   created dynamically. This behaves like a normal Python function when
   used as a method: the *self* argument will be inserted as the first
   positional argument, even before the *args* and *keywords* supplied to
   the :class:`partialmethod` constructor.

   Example::

      >>> class Cell:
      ...     def __init__(self):
      ...         self._alive = False
      ...     @property
      ...     def alive(self):
      ...         return self._alive
      ...     def set_state(self, state):
      ...         self._alive = bool(state)
      ...     set_alive = partialmethod(set_state, True)
      ...     set_dead = partialmethod(set_state, False)
      ...
```

```
    >>> c = Cell()
    >>> c.alive
    False
    >>> c.set_alive()
    >>> c.alive
    True

.. versionadded:: 3.4
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 379)

Unknown directive type "function".

```
.. function:: reduce(function, iterable[, initializer])

   Apply *function* of two arguments cumulatively to the items of *iterable*, from
   left to right, so as to reduce the iterable to a single value.  For example,
   ``reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])`` calculates ``((((1+2)+3)+4)+5)``.
   The left argument, *x*, is the accumulated value and the right argument, *y*, is
   the update value from the *iterable*.  If the optional *initializer* is present,
   it is placed before the items of the iterable in the calculation, and serves as
   a default when the iterable is empty.  If *initializer* is not given and
   *iterable* contains only one item, the first item is returned.

   Roughly equivalent to::

      def reduce(function, iterable, initializer=None):
          it = iter(iterable)
          if initializer is None:
              value = next(it)
          else:
              value = initializer
          for element in it:
              value = function(value, element)
          return value

   See :func:`itertools.accumulate` for an iterator that yields all intermediate
   values.
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 405)

Unknown directive type "decorator".

```
.. decorator:: singledispatch

   Transform a function into a :term:`single-dispatch <single
   dispatch>` :term:`generic function`.

   To define a generic function, decorate it with the ``@singledispatch``
   decorator. When defining a function using ``@singledispatch``, note that the
   dispatch happens on the type of the first argument::

     >>> from functools import singledispatch
     >>> @singledispatch
     ... def fun(arg, verbose=False):
     ...     if verbose:
     ...         print("Let me just say,", end=" ")
     ...     print(arg)

   To add overloaded implementations to the function, use the :func:`register`
   attribute of the generic function, which can be used as a decorator.  For
   functions annotated with types, the decorator will infer the type of the
   first argument automatically::

     >>> @fun.register
     ... def _(arg: int, verbose=False):
     ...     if verbose:
     ...         print("Strength in numbers, eh?", end=" ")
     ...     print(arg)
     ...
     >>> @fun.register
     ... def _(arg: list, verbose=False):
     ...     if verbose:
     ...         print("Enumerate this:")
     ...     for i, elem in enumerate(arg):
     ...         print(i, elem)
```

For code which doesn't use type annotations, the appropriate type argument can be passed explicitly to the decorator itself::

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

To enable registering :term:`lambdas<lambda>` and pre-existing functions, the :func:`register` attribute can also be used in a functional form::

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The :func:`register` attribute returns the undecorated function. This enables decorator stacking, :mod:`pickling<pickle>`, and the creation of unit tests for each variant independently::

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

When called, the generic function dispatches on the type of the first argument::

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with ``@singledispatch`` is registered for the base :class:`object` type, which means it is used if no better implementation is found.

If an implementation is registered to an :term:`abstract base class`, virtual subclasses of the base class will be dispatched to that implementation::

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

To check which implementation the generic function will choose for a given type, use the ``dispatch()`` attribute::

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

To access all registered implementations, use the read-only ``registry`` attribute::

```
    >>> fun.registry.keys()
    dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
              <class 'decimal.Decimal'>, <class 'list'>,
              <class 'float'>])
    >>> fun.registry[float]
    <function fun_num at 0x1035a2840>
    >>> fun.registry[object]
    <function fun at 0x103fe0000>

  .. versionadded:: 3.4

  .. versionchanged:: 3.7
     The :func:`register` attribute now supports using type annotations.
```

Transform a method into a :term:`single-dispatch <single dispatch>` :term:`generic function`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, **line 541**); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, **line 541**); *backlink*

Unknown interpreted text role "term".

To define a generic method, decorate it with the `@singledispatchmethod` decorator. When defining a function using `@singledispatchmethod`, note that the dispatch happens on the type of the first non-*self* or non-*cls* argument:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` supports nesting with other decorators such as :func:`@classmethod<classmethod>`. Note that to allow for `dispatcher.register`, `singledispatchmethod` must be the *outer most* decorator. Here is the `Negator` class with the `neg` methods bound to the class, rather than an instance of the class:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, **line 562**); *backlink*

Unknown interpreted text role "func".

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators: :func:`@staticmethod<staticmethod>`, :func:`@abstractmethod<abc.abstractmethod>`, and others.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, **line 584**); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 584); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 588)

Unknown directive type "versionadded".

```
.. versionadded:: 3.8
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 591)

Unknown directive type "function".

```
.. function:: update_wrapper(wrapper, wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATE

   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 ``__module__``, ``__name__``, ``__qualname__``, ``__annotations__``
   and ``__doc__``, the documentation string) and ``WRAPPER_UPDATES`` (which
   updates the wrapper function's ``__dict__``, i.e. the instance dictionary).

   To allow access to the original function for introspection and other purposes
   (e.g. bypassing a caching decorator such as :func:`lru_cache`), this function
   automatically adds a ``__wrapped__`` attribute to the wrapper that refers to
   the function being wrapped.

   The main intended use for this function is in :term:`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.

   :func:`update_wrapper` may be used with callables other than functions. Any
   attributes named in *assigned* or *updated* that are missing from the object
   being wrapped are ignored (i.e. this function will not attempt to set them
   on the wrapper function). :exc:`AttributeError` is still raised if the
   wrapper function itself is missing any attributes named in *updated*.

   .. versionadded:: 3.2
      Automatic addition of the ``__wrapped__`` attribute.

   .. versionadded:: 3.2
      Copying of the ``__annotations__`` attribute by default.

   .. versionchanged:: 3.2
      Missing attributes no longer trigger an :exc:`AttributeError`.

   .. versionchanged:: 3.4
      The ``__wrapped__`` attribute now always refers to the wrapped
      function, even if that function defined a ``__wrapped__`` attribute.
      (see :issue:`17482`)
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 635)

Unknown directive type "decorator".

```
.. decorator:: wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES)

   This is a convenience function for invoking :func:`update_wrapper` as a
   function decorator when defining a wrapper function.  It is equivalent to
   ``partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)``.
   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 :func:`example`
would have been lost.
```

:class:`partial` Objects

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 670); *backlink*

Unknown interpreted text role "class".

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 673); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 673); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 677)

Unknown directive type "attribute".

```
.. attribute:: partial.func

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 683)

Unknown directive type "attribute".

```
.. attribute:: partial.args

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 689)

Unknown directive type "attribute".

```
.. attribute:: partial.keywords

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

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 694); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 694); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 694); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 694); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]functools.rst`, line 694); *backlink*

Unknown interpreted text role "class".