miniutils Documentation

Release 1.0.1

scnerd

Oct 22, 2020

Contents:

CHAPTER 1

Progress Bars

Three progress bar utilities are provided, all leveraging the excellent tqdm library.

1.1 progbar

A simple iterable wrapper, much like the default `tqdm` wrapper. It can be used on any iterable to display a progress bar as it gets iterated:

```
for x in progbar(my_list):
    do_something_slow(x)
```

However, unlike the standard `tqdm` function, this code has two additional, useful behaviors: first, it automatically leverages the `ipywidgets` progress bar when run inside a jupyter notebook; second, if given an integer, it automatically creates `range(n)` to iterate on. Both of these features are available in the `tqdm` library, but as separate functions. `progbar` wraps them all into a single intuitive call. It even includes a `verbose` flag that can be disabled to eliminate the progress bar based on runtime variables, if so desired.

`miniutils.progress_bar.`**`progbar`**(*iterable*, **a*, *verbose=True*, ***kw*)

Prints a progress bar as the iterable is iterated over

Parameters

- **`iterable`** – The iterator to iterate over
- **`a`** – Arguments to get passed to tqdm (or tqdm_notebook, if in a Jupyter notebook)
- **`verbose`** – Whether or not to print the progress bar at all
- **`kw`** – Keyword arguments to get passed to tqdm

Returns The iterable that will report a progress bar

1.2 parallel_progbar

A parallel mapper based on `multiprocessing` that replaces `Pool.map`. In attempting to use `Pool.map`, I've had issues with unintuitive errors and, of course, wanting a progress bar of my map job's progress. Both of these are solved in `parallel_progbar`:

```
results = parallel_progbar(do_something_slow, my_list)
# Equivalent to a parallel version of [do_something_slow(x) for x in my_list]
```

This produces a pool of processes, and performs a map function in parallel on the items of the provided list.

Starmap behavior:

```
results = parallel_progbar(do_something_slow, my_list, starmap=True)
# [do_something_slow(*x) for x in my_list]
```

And/or flatmap behavior:

```
results = parallel_progbar(make_more_things, my_things, flatmap=True)
# Equivalent to a parallel version of [y for x in my_things for y in make_more_
↪things(x)]
```

It also supports runtime disabling, limited number of parallel processes, shuffling before mapping (in case the order of your list puts, say, a few slowest items near the end), and even an optional second progress bar when performing a flatmap. This second bar just reports the number of items output (`y` in the case above), while the main progress bar counts down the number of finished inputs (`x`).

`miniutils.progress_bar.`**`parallel_progbar`**(**args*, ***kwargs*)

Performs a parallel mapping of the given iterable, reporting a progress bar as values get returned

Parameters

- **`mapper`** – The mapping function to apply to elements of the iterable
- **`iterable`** – The iterable to map
- **`nprocs`** – The number of processes (defaults to the number of cpu's)
- **`starmap`** – If true, the iterable is expected to contain tuples and the mapper function gets each element of a tuple as an argument
- **`flatmap`** – If true, flatten out the returned values if the mapper function returns a list of objects
- **`shuffle`** – If true, randomly sort the elements before processing them. This might help provide more uniform runtimes if processing different objects takes different amounts of time.
- **`verbose`** – Whether or not to print the progress bar
- **`verbose_flatmap`** – If performing a flatmap, whether or not to report each object as it's returned
- **`timeout`** – The number of seconds to wait for each worker process after completing
- **`kwargs`** – Any other keyword arguments to pass to the progress bar (see `progbar`)

Returns A list of the returned objects, in the same order as provided

1.3 iparallel_progbar

This has the exact same behavior as `parallel_progbar`, but produces an unordered generator instead of a list, yielding results as soon as they're available. It also permits a `max_cache` argument that allows you to limit the number of computed results available to the generator.

```
for result in iparallel_progbar(do_something_slow, my_list):
    print("Result {} done!".format(result))
```

`miniutils.progress_bar.`**`iparallel_progbar`**`(*args, **kwargs)`

Performs a parallel mapping of the given iterable, reporting a progress bar as values get returned. Yields objects as soon as they're computed, but does not guarantee that they'll be in the correct order.

Parameters

- **`mapper`** – The mapping function to apply to elements of the iterable
- **`iterable`** – The iterable to map
- **`nprocs`** – The number of processes (defaults to the number of cpu's)
- **`starmap`** – If true, the iterable is expected to contain tuples and the mapper function gets each element of a tuple as an argument
- **`flatmap`** – If true, flatten out the returned values if the mapper function returns a list of objects
- **`shuffle`** – If true, randomly sort the elements before processing them. This might help provide more uniform runtimes if processing different objects takes different amounts of time.
- **`verbose`** – Whether or not to print the progress bar
- **`verbose_flatmap`** – If performing a flatmap, whether or not to report each object as it's returned
- **`max_cache`** – Maximum number of mapped objects to permit in the queue at once
- **`timeout`** – The number of seconds to wait for each worker process after completing
- **`kwargs`** – Any other keyword arguments to pass to the progress bar (see `progbar`)

Returns A list of the returned objects, in whatever order they're done being computed

CHAPTER 2

Property Cache

2.1 Basic Property

In some cases, an object has properties that don't need to be computed until necessary, and once computed are generally static and could just be cached. This could be accomplished using the following simple recipe:

```
class Obj:
    def __init__(self):
        self._attribute = None
        ...

    @property
    def attribute(self):
        if self._attribute is None:
            self._attribute = some_slow_computation(self)
        return self._attribute
```

If you want to support re-computation (besides just setting the object to `None` again), it's not hard to add:

```
class Obj:
    def __init__(self):
        self._attribute = None
        self._need_attribute = True
        ...

    @property
    def attribute(self):
        if self._need_attribute:
            self._attribute = some_slow_computation(self)
            self._need_attribute = False
        return self._attribute


...
attr1 = my_obj.attribute
```

(continues on next page)

(continued from previous page)

```
my_obj._need_attribute = True
attr2 = my_obj.attribute  # Re-computes attribute
```

Adding inter-dependence between such properties is not hard, but quickly becomes verbose. In fact, all of this code is verbose relative to the simple goal: for some property `x`, define its value, but don't actually compute it until necessary, and allow the code to make it "necessary" again. This is easy to describe, and easy to think of, but just convoluted to code (but fortunately, easy to template).

To simplify this process, `miniutils` provides a `CachedProperty` decorator that's simple by default, and moderately powerful when necessary. Let's take a look at a simple use case first, then we'll examine its capabilities:

```
class Obj:
    @CachedProperty()
    def attribute(self):
        return some_slow_computation(self)
```

That's all you need. No need to initialize, set up flags, or anything. It's all handled automatically. A use case like above might look like:

```
attr1 = my_object.attribute  # Computed the first time
attr2 = my_object.attribute  # Loaded from cache
assert attr1 is attr2
del my_object.attribute  # Deletes the cached object and marks for re-computation
attr3 = my_object.attribute  # Re-computes the value
```

Despite being simple to use, it's still a fairly powerful decorator:

- Like `@property`, this method is converted to a property (in fact, the `property` function is used under the hood, so you don't have any `CachedProperty` objects floating around)
- The result is lazy-computed, just like you'd expect from a property
- The result is cached and returned instantly if not marked for re-computation (note that the object doesn't have to be hashable since there's no lookup being performed)
- Its computation can affect the computation of other properties, and thus automatically mark those properties for re-computation when needed (i.e., it maintains a dependency chain amongst CachedProperties)
- A simple setter can be automatically defined which invalidates downstream properties without needing more code (note that, at this time, you can't safely define a custom setter, you can either use the default or let the property be unsettable)
- If the property returns a basic iterable (list, dictionary, set), it's wrapped so that modifications to its content (if permitted) invalidate downstream properties.

A key feature not yet demonstrated is the ability to add dependencies amongst properties. Essentially, this defines a directed graph where resetting, re-computing, or altering upstream properties marks all dependent downstream properties for re-computation. This can be seen in the following demonstration:

```
class Printer:
    @CachedProperty('b', settable=True)
    def a(self):
        print("Running a")
        return 5

    @CachedProperty('c', is_collection=True)
    def b(self):
        print("Running b")
```

(continues on next page)

(continued from previous page)

```
        return [self.a] * 100

    @CachedProperty('d')
    def c(self):
        print("Running c")
        return sum(self.b)

    @CachedProperty()
    def d(self):
        print("Running d")
        return str(self.c ** 2)

p = Printer()
p.a          # Computes A
p.c          # Computes C, during which it computes B
p.a = 3      # Sets A, invalidating B and C (and D, if it weren't already invalid)
p.c          # Computes C, and thus B, again
p.c          # Returns the cached value for C
p.b[0] = 0   # Alters a value within B (not B itself), which correctly invalidates C
p.c          # Computes C, using cached B
del p.a      # Invalidates A, and therefore B and C
p.d          # Computes D, and thus C, B, and A
```

This isn't the complete feature set of the decorator, but it's a good initial taste of what can be accomplished using it.

class miniutils.caching.**CachedProperty**(**affects*, *settable=False*, *threadsafe=True*, *is_collection=False*, *allow_collection_mutation=True*)

Marks this property to be cached. Delete this property to remove the cached value and force it to be rerun.

Parameters

- **affects** – Strings that list the names of the other properties in this class that are directly invalidated when this property's value is altered
- **settable** – Whether or not to allow this property to have values assigned directly to it
- **threadsafe** – Whether or not to restrict execution of this property's code to a single thread at a time (safe for recursive calls)
- **is_collection** – Whether or not this property returns a collection (currently supports lists, sets, and dictionaries; others might not work exactly as expected)
- **allow_collection_mutation** – Whether or not the returned collection should allow its values to be altered

2.2 Indexed Property

Even using the above tools, it is non-concise to allow indexing into a property where values are lazily computed.

The `LazyDictionary` decorator allows you to write a `__getitem__` style property that can be used like a dictionary and has its results cached:

```
class Primes:
    @LazyDictionary()
    def is_prime(self, i):
        if not isinstance(i, int) or i < 1:
```

(continues on next page)

(continued from previous page)

```
            raise ValueError("Can only check if a positive integer is prime")
        elif i in [1, 2]:
            return True
        elif i % 2 == 0:
            return False
        else:
            return all(i % p != 0 for p in range(3, int(math.sqrt(i)) + 1, 2) if self.
↪is_prime[p])

p = Primes()
p.is_prime[5] # True, caches the fact that 1, 2, and 3 are prime
p.is_prime[500] # False, caches all primes up to sqrt(500)
p.is_prime[501] # False, virtually instant since it uses the cached primes used to␣
↪compute is_prime[500]
```

The indexing notation is used and preferred to make clear that this decorator only aims to support one hashable argument, and is meant to behave like a dictionary or list. It is not iterable, since the result of that would depend on whatever prior code happened to be executed. Instead, you should iterate through all desired keys, and simply index them; that way, any that need to be re-computed are, and those that can are loaded from cache.

This plugs cleanly into `CachedProperty`, accepting a list of properties whose values are invalidated when this dictionary is modified. It also supports allowing or disallowing explicit assignment to certain indices:

```
p = Primes()
p.is_prime[3] = False
p.is_prime[9] # This is now True, since there is no lesser known prime
```

This is meant to provide a slight additional feature to having a cached dictionary, though honestly it's probably a very small improvement over `self.is_prime = defaultdict(self._is_prime)`, since it has the additions of invalidating cached properties and making values dependant on their indices.

Values can be explicitly assigned to indices (if `allow_collection_mutation=True`); assigned values override cached values. Raised `KeyError``s are cached to prevent re-running indices where failure is known. If an error is not due solely to the index, raise some other error to allow that index to be retried later if some variation to the program's state might allow it to succeed. ``.get(key, default)` and `.update(dict)` are also provided to offer a more dictionary-like interface. A particular object instance will have a `miniutils.caching._LazyDictionary` instance which provides its caching, though the decorated function is once again replaced with a simple `@property`.

class miniutils.caching.**LazyDictionary**(**affects*, *allow_collection_mutation=False*)

Marks this indexable property to be a cached dictionary. Delete this property to remove the cached value and force it to be rerun.

Parameters

- **affects** – Strings that list the names of the other properties in this class that are directly invalidated when this property's value is altered
- **allow_collection_mutation** – Whether or not the returned collection should allow its values to be altered

2.3 File-backed Function Cache

As a file-based alternative to simple function caching (such as that provided by `functools.lru_cache`), *miniutils.caching.FileCached* provides caching of a function's results using `shelve` as its storage back-

end. This is primarily intended for long-run file processing scripts, and as such it natively supports invalidating cache items if relied-upon files are modified since when the cache entry was created.

There are several ways to use this cache. The simplest is to use it as a decorator, leveraging *miniutils.caching.file_cached_decorator()*. The following example stores the results of `load_data` in a cache at `./preprocessed`, which gets automatically invalidated when `/path/to/data.csv` gets modified:

```
@file_cached_decorator('./preprocessed', files_used=['/path/to/data.csv'])
def load_data():
    df = pandas.read_csv('/path/to/data.csv')
    # Modify, clean, process data
    return df
```

This could also be accomplished on a function not defined in the user code, using *miniutils.caching.FileCached* directly:

```
data = FileCached(load_data, './preprocessed', files_used=['/path/to/data.csv'])
```

By offloading the generation of the cache to the caller code, it's also possible to dynamically provide the list of files being used when they are arguments to the function:

```
def load_data(path):
    df = pandas.read_csv(path)
    # ...

data = FileCached(load_data, './preprocessed', files_used=[data_path])(data_path)
```

This use of *miniutils.caching.FileCached* is how it is meant to be used when attempting to store function results across multiple runs of a script. Each time the script is run, it will connect to the same persistent on-disk cache, update if function arguments or relied-upon files change, and synchronize any new function results back to disk before the program exits.

By default, *miniutils.caching.FileCached* and its decorator form generate a cache filepath based on the function's name if no explicit name is set. It is recommended not to use this default name if you wish to use the cache between runs of Python, since any change to the function's name will invalidate the cache; also, this breaks if you wish to cache multiple functions with the same name.

Warning: Note that `shelve`, and therefore *miniutils.caching.FileCached*, is not thread-safe or multiprocess-safe, so this cache will likely fail if being used in any parallel fashion. To use a data store in a parallel fashion, you should probably rely on a robust database system of some sort, such as MongoDB.

Warning: When purging a file cache, *miniutils.caching.FileCached* deletes all files matching its database's filepath. Make sure that the file path given for the cache has no relation to any other code or data files used by your program.

class miniutils.caching.**FileCached**(*fn*, *cache_path=None*, *files_used=None*, *auto_purge=False*)

Caches function results to a file to save re-computation of highly expensive calls

Parameters

- **fn** (*function*) – The functions whose result should be cached
- **cache_path** (*str*) – No-extension file path where cache should be kept

- **files_used** (*Iterable*) – List of files that could effect the result of this function; cache results are invalidated if any of these files are updated since the last function call
- **auto_purge** – If True, deletes the file cache when this cache object passes out of scope

Type auto_purge: bool

cache_clear(*create_new_shelf=True*)
Deletes the underlying cache

cache_info()
Gets information about this cache.

Returns A named tuple containing the number of cache `hits` and `misses`

miniutils.caching.**file_cached_decorator**(**args*, ***kwargs*)
A decorator version of `FileCached`

Parameters

- **cache_path** (*str*) – No-extension file path where cache should be kept
- **files_used** (*Iterable*) – List of files that could effect the result of this function; cache results are invalidated if any of these files are updated since the last function call
- **auto_purge** – If True, deletes the file cache when this cache object passes out of scope

Type auto_purge: bool

Returns A decorator for a function

Return type function

CHAPTER 3

Nesting Python 2

In *very* rare situations, the standard means of Python2 compatibility within Python3 (such as `six`, `2to3`, or `__futures__`) might simply be insufficient. Sometimes, you just need to run Python2 wholesale to get the correct behavior.

> This is not generally advised at all. I built this out of necessity, where identical function calls to a built-in Python package worked in Python2 and broke in Python3, and I could see no other way to solve the problem. Please exhaust all other options before deciding to use this hack.

In the vein of making complex modules in support of simple code, I wrapped the entire behavior into a function decorator. Define the function you want to run in Python2, decorate it, then just run it like you normally would. Voila, it's executed in a Python2 subprocess.

This works essentially using code templating. A Python2 instance is kicked off as a subprocess; it loads the parameters needed to run the function (as given to the decorator); finally, it sits in an infinite loop receiving arguments as pickles, running them through the function, and returning the results as pickles. It's designed to run self-contained functions, with some support for wrapping functions defined in external modules (though generally, in this case, I'd recommend writing a simple self-contained function that loads that module and runs the function).

Let's take a look at a minimal example:

```
@MakePython2()
def get_version():
    import sys
    return sys.version_info[0]

get_version()  # Reports that we're in Python 2

import sys
sys.version_info[0]  # Reports that we're in Python 3
```

Of course, not every function is self-contained like this. To handle the majority of easy cases, the `MakePython2` decorator supports pre-defining a set of imports and global variables.

Imports are given as a list of items, each of which should be either a simple string:

```
@MakePython2(imports=['sys'])
def get_version():
    return sys.version_info[0]
```

or as a tuple of `(package, name)`:

```
@MakePython2(imports=[('sys', 'another_name')])
def get_version():
    return another_name.version_info[0]
```

Global variables (if they can be pickled using protocol 2, the highest protocol for Python2) can be given as a dictionary of `dict(name=value, ...)`:

```
@MakePython2(global_values={'x': 5})
def add(y):
    return x + y
```

Additional features include changing the Python2 executable path, specifying that the function code shouldn't be copied to the Python2 instance (e.g., if you're just running a single function from an external module), and specifying the function to execute by name instead of by passing the function directly.

For example, to execute an external function, you can use the class as a wrapper instead of using the decorator notation:

```
uname = MakePython2('os.uname', imports=['os'], copy_function_body=False).function
```

class miniutils.py2_wrap.**MakePython2**(*func=None*, ***, *imports=None*, *global_values=None*, *copy_function_body=True*, *python2_path='python2'*)

Make a function execute within a Python 2 instance

Parameters

- **func** – The function to wrap. If not specified, this class instance behaves like a decorator
- **imports** – Any import statements the function requires. Should be a list, where each element is either a string (e.g., `'sys'` for `import sys`) or a tuple (e.g., `('os.path', 'path')` for `import os.path as pas`)
- **global_values** – A dictionary of global variables the function relies on. Key must be strings, and values must be picklable
- **copy_function_body** – Whether or not to copy the function's source code into the Python 2 instance
- **python2_path** – The path to the Python 2 executable to use

__init__(*func=None*, ***, *imports=None*, *global_values=None*, *copy_function_body=True*, *python2_path='python2'*)

Make a function execute within a Python 2 instance

Parameters

- **func** – The function to wrap. If not specified, this class instance behaves like a decorator
- **imports** – Any import statements the function requires. Should be a list, where each element is either a string (e.g., `'sys'` for `import sys`) or a tuple (e.g., `('os.path', 'path')` for `import os.path as pas`)
- **global_values** – A dictionary of global variables the function relies on. Key must be strings, and values must be picklable
- **copy_function_body** – Whether or not to copy the function's source code into the Python 2 instance

- **python2_path** – The path to the Python 2 executable to use

CHAPTER 4

Miscellaneous

4.1 Code Contracts

Code contracting seems like a great way to define and document your code's expected behavior, easily integrate bounds checking, and just generally write code that tries to avoid bugs. The pycontracts package provides this capability within python, but as soon as I started using it I realized that it was meant primarily to be robust, not concise. For example, consider the following code:

```
class ObjA:
    pass

class ObjB:
    pass

@contract
def sample_func(a):
    """A function that requires an A object

    :param a: A thing
    :type a: ObjA
    :return: What you gave it
    :rtype: ObjB
    """
    return ObjB()
```

This seems intuitive what should happen–you're not using any complex attributes of the types, merely indicating that is should be of that type–but `pycontracts` will croak on this because you haven't explicitly told it about your two new types.

`miniutils.magic_contract` is a little wrapper around the `contract` decorator that looks through the function's local namespace, finds types that aren't already registered with `pycontracts`, and adds them as a simple `isinstance` check. Using it, we can write almost the exact same code:

```
class ObjA:
    pass


class ObjB:
    pass


@magic_contract  # Uses the magic contract
def sample_func(a):
    """A function that requires an A object

    :param a: A thing
    :type a: ObjA
    :return: What you gave it
    :rtype: ObjB
    """
    return ObjB()
```

And now the function works like you'd expect. If you want to do something more complex when adding an object as a contractable type, just use `contracts.new_contract` like you normally would, and `magic_contract` won't clobber your definition. Also, since this decorator is just a wrapper around `contracts.contract`, you can continue using `pycontracts` as always, and the magic contract won't affect any of the rest of your code.

`miniutils.magic_contract.`**`magic_contract`**`(`*args*, ***kwargs*`)`

Drop-in replacement for `pycontracts.contract` decorator, except that it supports locally-visible types

Parameters

- **`args`** – Arguments to pass to the `contract` decorator
- **`kwargs`** – Keyword arguments to pass to the `contract` decorator

Returns The contracted function

4.2 Simplifying Decorators

When writing a decorator that could be used like `@deco` or `@deco()`, there's a little code I've found necessary in order to make both cases function identically. I've isolated this code into another decorator (meta-decorator?) to keep my other decorators simple (since, let's be honest, decorators are usually convoluted enough as is).

Consider the following decorator definition:

```
def deco(return_name=False):
    def inner_deco(func):
        def inner(*a, **kw):
            if return_name:
                return func.__name__, func(*a, **kw)
            else:
                return func(*a, **kw)
        return inner
    return inner_deco

@deco()  # Works correctly
def g(i):
    return i
```

(continues on next page)

(continued from previous page)

```
@deco(True)  # Works correctly
def h(i):
    return i

@deco(return_name=True)  # Works correctly
def k(i):
    return i

@deco  # Fails, since f gets assigned to return_names instead of func
def f(i):
    return i
```

This makes sense, but is somewhat annoying when parameters aren't required, such as is the case in several built-in Python decorators. To make this last case work like the first, we can simply decorate our decorator:

```
@optional_argument_decorator
def deco(return_name=False):
    def inner_deco(func):
        def inner(*a, **kw):
            if return_name:
                return func.__name__, func(*a, **kw)
            else:
                return func(*a, **kw)
        return inner
    return inner_deco

@deco()  # Works correctly
def g(i):
    return i

@deco(True)  # This still works
def h(i):
    return i

@deco(return_name=True)  # As does this
def k(i):
    return i

@deco  # Now this works too!
def f(i):
    return i
```

miniutils.opt_decorator.**optional_argument_decorator**(*_decorator*)

Decorate your decorator with this to allow it to always receive *args and **kwargs, making @deco equivalent to @deco()

4.3 Logging Made Easy

The standard `logging` module provides a lot of great functionality, but there are a few simplifications missing:

1. Intuitive colored logging to terminal
2. Fallback logging utilities when "logging" should only be enabled in certain contexts
3. "One-click" logging setup

As a slight simplification, `miniutils` provides a wrapper around the `logging` module to provide these features.

4.3.1 Usage

To use the logging features listed below, just import the logger:

```
from miniutils.logs import logger
```

If you want to use logging when available, but fall back to simply `print` to `stderr` when the logger isn't initialized elsewhere (for example, if you're writing a helper module that shouldn't dictate the logging format used in the user code), you can obtain a proxy logger object:

```
from miniutils import logs_base as logger
```

This module has `info`, `warn`, `warning`, `error`, `critical`, and `log` calls that use the logger when available, or fall back to a simple `print` statement otherwise. If the logger gets loaded from `miniutils.logs` later, these calls get swapped out automatically for their full-featured logger alternatives.

To change the logger's configuration, do something like the following:

```
from miniutils.logs import enable_logging
enable_logging(fmt_str='$(asctime) ( %(levelname) ) - $(message)')
```

This will swap out the logger and handlers that the rest of the logging utilities use.

`miniutils.logs.`**`enable_logging`**(*log_level='NOTSET'*, ***, *logdir=None*, *use_colors=True*, *capture_warnings=True*, *format_str='%(asctime)s [%(launch_script)s | %(levelname)s]: %(message)s'*)

4.3.2 Colored Logging

The `coloredlogs` module didn't quite work as expected when I tried to use it. It provides lots of handles and controls, but wasn't quite as intuitive as I expected it to be. To provide this more intuitive functionality, I wrap `coloredlogs` with a custom formatter that behaves more like expected:

- Don't assume the foreground color (it assumes black-on-white by default; I switch this to pulling the foreground color from the currently active color swatch)
- Uses case-sensitive match for level names (e.g., 'DEBUG', 'INFO', etc.), which seems silly. I monkey-patch this to be case insensitive
- Doesn't color aliases properly, even though it nominally supports name aliases

4.4 Timing

Simple `printf`-like timing utilities when proper profiling won't quite work.

4.4.1 Timing Functions

To make a timed call to a function:

```
from time import sleep
from miniutils.timing import timed_call

def f(a, *, x=1, sleep_dur=0.1):
    sleep(sleep_dur)
    return a * x

result = timed_call(f, 2, x=3, sleep_dur=0.11)
# "Call to 'f' took 0.110240s"
```

To make all calls to a function timed:

```
from time import sleep
from miniutils.timing import make_timed

@make_timed
def g(a, *, x=1, sleep_dur=0.1):
    sleep(sleep_dur)
    return a * x

g(2, x=3, sleep_dur=0.11)
# "Call to 'g' took 0.110242s"
```

miniutils.timing.**timed_call**(*func*, **args*, *log_level='DEBUG'*, ***kwargs*)

Logs a function's run time

Parameters

- **func** – The function to run
- **args** – The args to pass to the function
- **kwargs** – The keyword args to pass to the function
- **log_level** – The log level at which to print the run time

Returns The function's return value

miniutils.timing.**make_timed**(*func*)

A decorator to make a function print its execution time whenever it gets called

4.4.2 Timing Blocks

Use `tic`/`toc` to time and report the run times of different chunks of code:

```
from time import sleep
from miniutils.timing import tic

toc = tic()  # Just marks start time
sleep(0.2)
toc('Slept for 0.2 seconds')
# "sample_timing.py:6 - Slept for 0.2 seconds - 0.200329s (total=0.2s)"
sleep(.1)
toc('Slept for 0.1 seconds')
# "sample_timing.py:8 - Slept for 0.1 seconds - 0.100217s (total=0.3s)"
```

This utility is just less verbose than tracking various times yourself. The output is printed to the log for later review. It can also accept a custom print format string, including information about the code calling `toc()` and runtimes since the last `tic`/`toc`.

miniutils.timing.**tic**(*log_level='DEBUG'*, *fmt='{file}:{line} - {message} - {diff:0.6f}s (total={total:0.1f}s)'*, *verbose=True*)

A minimalistic `printf`-type timing utility. Call this function to start timing individual sections of code

Parameters

- **log_level** – The level at which to log block run times
- **fmt** – The format string to use when logging times. Available arguments include:
  - file, line, func, code_text: The stack frame information which called this timer
  - diff: The time since the last timer printout was called
  - total: The time since this timing block was started
  - message: The message passed to this timing printout
- **verbose** – If False, suppress printing messages

Returns A function that reports run times when called

CHAPTER 5

API

5.1 Caching

class miniutils.caching.**CachedProperty**(**affects*, *settable=False*, *threadsafe=True*, *is_collection=False*, *allow_collection_mutation=True*)

Marks this property to be cached. Delete this property to remove the cached value and force it to be rerun.

Parameters

- **affects** – Strings that list the names of the other properties in this class that are directly invalidated when this property's value is altered
- **settable** – Whether or not to allow this property to have values assigned directly to it
- **threadsafe** – Whether or not to restrict execution of this property's code to a single thread at a time (safe for recursive calls)
- **is_collection** – Whether or not this property returns a collection (currently supports lists, sets, and dictionaries; others might not work exactly as expected)
- **allow_collection_mutation** – Whether or not the returned collection should allow its values to be altered

__init__(**affects*, *settable=False*, *threadsafe=True*, *is_collection=False*, *allow_collection_mutation=True*)

Marks this property to be cached. Delete this property to remove the cached value and force it to be rerun.

Parameters

- **affects** – Strings that list the names of the other properties in this class that are directly invalidated when this property's value is altered
- **settable** – Whether or not to allow this property to have values assigned directly to it
- **threadsafe** – Whether or not to restrict execution of this property's code to a single thread at a time (safe for recursive calls)

- **is_collection** – Whether or not this property returns a collection (currently supports lists, sets, and dictionaries; others might not work exactly as expected)
- **allow_collection_mutation** – Whether or not the returned collection should allow its values to be altered

5.2 Progress Bar

miniutils.progress_bar.**progbar**(*iterable*, **a*, *verbose=True*, ***kw*)

Prints a progress bar as the iterable is iterated over

Parameters

- **iterable** – The iterator to iterate over
- **a** – Arguments to get passed to tqdm (or tqdm_notebook, if in a Jupyter notebook)
- **verbose** – Whether or not to print the progress bar at all
- **kw** – Keyword arguments to get passed to tqdm

Returns The iterable that will report a progress bar

miniutils.progress_bar.**parallel_progbar**(**args*, ***kwargs*)

Performs a parallel mapping of the given iterable, reporting a progress bar as values get returned

Parameters

- **mapper** – The mapping function to apply to elements of the iterable
- **iterable** – The iterable to map
- **nprocs** – The number of processes (defaults to the number of cpu's)
- **starmap** – If true, the iterable is expected to contain tuples and the mapper function gets each element of a tuple as an argument
- **flatmap** – If true, flatten out the returned values if the mapper function returns a list of objects
- **shuffle** – If true, randomly sort the elements before processing them. This might help provide more uniform runtimes if processing different objects takes different amounts of time.
- **verbose** – Whether or not to print the progress bar
- **verbose_flatmap** – If performing a flatmap, whether or not to report each object as it's returned
- **timeout** – The number of seconds to wait for each worker process after completing
- **kwargs** – Any other keyword arguments to pass to the progress bar (see `progbar`)

Returns A list of the returned objects, in the same order as provided

miniutils.progress_bar.**iparallel_progbar**(**args*, ***kwargs*)

Performs a parallel mapping of the given iterable, reporting a progress bar as values get returned. Yields objects as soon as they're computed, but does not guarantee that they'll be in the correct order.

Parameters

- **mapper** – The mapping function to apply to elements of the iterable
- **iterable** – The iterable to map

- **nprocs** – The number of processes (defaults to the number of cpu's)
- **starmap** – If true, the iterable is expected to contain tuples and the mapper function gets each element of a tuple as an argument
- **flatmap** – If true, flatten out the returned values if the mapper function returns a list of objects
- **shuffle** – If true, randomly sort the elements before processing them. This might help provide more uniform runtimes if processing different objects takes different amounts of time.
- **verbose** – Whether or not to print the progress bar
- **verbose_flatmap** – If performing a flatmap, whether or not to report each object as it's returned
- **max_cache** – Maximum number of mapped objects to permit in the queue at once
- **timeout** – The number of seconds to wait for each worker process after completing
- **kwargs** – Any other keyword arguments to pass to the progress bar (see `progbar`)

Returns A list of the returned objects, in whatever order they're done being computed

5.3 Python 2

class miniutils.py2_wrap.**MakePython2**(*func=None*, ***, *imports=None*, *global_values=None*, *copy_function_body=True*, *python2_path='python2'*)

Make a function execute within a Python 2 instance

Parameters

- **func** – The function to wrap. If not specified, this class instance behaves like a decorator
- **imports** – Any import statements the function requires. Should be a list, where each element is either a string (e.g., `'sys'` for `import sys`) or a tuple (e.g., `('os.path', 'path')` for `import os.path as pas`)
- **global_values** – A dictionary of global variables the function relies on. Key must be strings, and values must be picklable
- **copy_function_body** – Whether or not to copy the function's source code into the Python 2 instance
- **python2_path** – The path to the Python 2 executable to use

__init__(*func=None*, ***, *imports=None*, *global_values=None*, *copy_function_body=True*, *python2_path='python2'*)

Make a function execute within a Python 2 instance

Parameters

- **func** – The function to wrap. If not specified, this class instance behaves like a decorator
- **imports** – Any import statements the function requires. Should be a list, where each element is either a string (e.g., `'sys'` for `import sys`) or a tuple (e.g., `('os.path', 'path')` for `import os.path as pas`)
- **global_values** – A dictionary of global variables the function relies on. Key must be strings, and values must be picklable

- **copy_function_body** – Whether or not to copy the function's source code into the Python 2 instance
- **python2_path** – The path to the Python 2 executable to use

5.4 Pragma

5.5 Miscellaneous

5.5.1 Magic Contracting

miniutils.magic_contract.**magic_contract**(**args*, ***kwargs*)

Drop-in replacement for `pycontracts.contract` decorator, except that it supports locally-visible types

Parameters

- **args** – Arguments to pass to the `contract` decorator
- **kwargs** – Keyword arguments to pass to the `contract` decorator

Returns The contracted function

5.5.2 Simplifying Decorators

miniutils.opt_decorator.**optional_argument_decorator**(*_decorator*)

Decorate your decorator with this to allow it to always receive *args and **kwargs, making @deco equivalent to @deco()

5.5.3 Logging

miniutils.logs.**enable_logging**(*log_level='NOTSET'*, ***, *logdir=None*, *use_colors=True*, *capture_warnings=True*, *format_str='%(asctime)s [%(launch_script)s | %(levelname)s]: %(message)s'*)

5.5.4 Timing

miniutils.timing.**timed_call**(*func*, **args*, *log_level='DEBUG'*, ***kwargs*)

Logs a function's run time

Parameters

- **func** – The function to run
- **args** – The args to pass to the function
- **kwargs** – The keyword args to pass to the function
- **log_level** – The log level at which to print the run time

Returns The function's return value

miniutils.timing.**make_timed**(*func*)

A decorator to make a function print its execution time whenever it gets called

miniutils.timing.**tic**(*log_level='DEBUG'*, *fmt='{file}:{line} - {message} - {diff:0.6f}s (total={total:0.1f}s)'*, *verbose=True*)

A minimalistic `printf`-type timing utility. Call this function to start timing individual sections of code

Parameters

- **log_level** – The level at which to log block run times
- **fmt** – The format string to use when logging times. Available arguments include:
  - file, line, func, code_text: The stack frame information which called this timer
  - diff: The time since the last timer printout was called
  - total: The time since this timing block was started
  - message: The message passed to this timing printout
- **verbose** – If False, suppress printing messages

Returns A function that reports run times when called

CHAPTER 6

Overview

This module provides numerous helper utilities for Python3.X code to add functionality with minimal code footprint. It has tools for the following tasks:

- Progress bars on serial loops and parallel mappings (leveraging the excellent `tqdm` library)
- Simple lazy-compute and caching of class properties, including dependency chaining
- Executing Python2 code from within a Python3 program
- More intuitive contract decorator (leveraging `pycontracts`)

CHAPTER 7

Installation

As usual, you can install the latest code version directly from Github:

```
pip install git+https://github.com/scnerd/miniutils
```

Or you can `pip` install the latest release from PyPi:

```
pip install miniutils
```

CHAPTER 8

Examples

To get started, you can import your desired utilities directly from `miniutils`. For example, to use the `CachedProperty` decorator:

```
from miniutils import CachedProperty

class MyClass:
   @CachedProperty
   def attribute(self):
      return some_slow_computation(self)
```

Or to use the progress bar utilities:

```
from miniutils import progbar, parallel_progbar

def mapper(x):
   return x**2

assert [mapper(i) for i in progbar(100)] == parallel_progbar(mapper, range(100))
```

To see documentation for each feature, look through this documentation or the table of contents above.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

Symbols

C

E

F

I

L

M

O

P

T