bitmath Documentation

Release 1.3.1

Tim Bielawa

December 06, 2016

Contents

bitmath simplifies many facets of interacting with file sizes in various units. Originally focusing on file size unit conversion, functionality now includes:

- Converting between **SI** and **NIST** prefix units (`kB` to `GiB`)
- Converting between units of the same type (SI to SI, or NIST to NIST)
- Automatic human-readable prefix selection (like in hurry.filesize)
- Basic arithmetic operations (subtracting 42KiB from 50GiB)
- Rich comparison operations (`1024 Bytes == 1KiB`)
- bitwise operations (<<, >>, `&`, `|`, `^`)
- Reading a device's storage capacity (Linux/OS X support only)
- argparse integration as a custom type
- progressbar integration as a better file transfer speed widget
- String parsing
- Sorting

In addition to the conversion and math operations, *bitmath* provides human readable representations of values which are suitable for use in interactive shells as well as larger scripts and applications. The format produced for these representations is customizable via the functionality included in stdlibs string.format.

In discussion we will refer to the NIST units primarily. I.e., instead of "megabyte" we will refer to "mebibyte". The former is `10^3 = 1,000,000` bytes, whereas the second is `2^20 = 1,048,576` bytes. When you see file sizes or transfer rates in your web browser, most of the time what you're really seeing are the base-2 sizes/rates.

Don't Forget! The source for bitmath is available on GitHub.

And did we mention there's almost 200 unittests? Check them out for yourself.

- *Examples* after the TOC.

CHAPTER 1

Installation

The easiest way to install bitmath is via `dnf` (or `yum`) if you're on a Fedora/RHEL based distribution. bitmath is available in the main Fedora repositories, as well as the EPEL6 and EPEL7 repositories. There are now dual python2.x and python3.x releases available.

Python 2.x:

```
$ sudo dnf install python2-bitmath
```

Python 3.x:

```
$ sudo dnf install python3-bitmath
```

Note: Upgrading: If you have the old *python-bitmath* package installed presently, you could also run `sudo dnf update python-bitmath` instead

PyPi:

You could also install bitmath from PyPi if you like:

```
$ sudo pip install bitmath
```

Note: **pip** installs need pip >= 1.1. To workaround this, download bitmath, from PyPi and then `pip install bitmath-x.y.z.tar.gz`. See issue #57 for more information.

PPA:

Ubuntu Xenial, Wily, Vivid, Trusty, and Precise users can install bitmath from the launchpad PPA:

```
$ sudo add-apt-repository ppa:tbielawa/bitmath
$ sudo apt-get update
$ sudo apt-get install python-bitmath
```

Source:

Or, if you want to install from source:

```
$ sudo python ./setup.py install
```

If you want the bitmath manpage installed as well:

```
$ sudo make install
```

CHAPTER 2

Contents

2.1 The `bitmath` Module

2.1.1 Functions

This section describes utility functions included in the *`bitmath`* module.

bitmath.getsize()

`bitmath.`**`getsize`**(*path*[, *bestprefix=True*[, *system=NIST*]])

Return a bitmath instance representing the size of a file at any given path.

Parameters

- **`path`** (*`string`*) – The path of a file to read the size of
- **`bestprefix`** (*`bool`*) – **Default:** `True`, the returned instance will be in the best human-readable prefix unit. If set to `False` the result is a `bitmath.Byte` instance.
- **`system`** (One of *`bitmath.NIST`* or *`bitmath.SI`*) – **Default:** *`bitmath.NIST`*. The preferred system of units for the returned instance.

Internally *`bitmath.getsize()`* calls `os.path.realpath()` before calling `os.path.getsize()` on any paths.

Here's an example of where we'll run *bitmath.getsize()* on the bitmath source code using the defaults for the `bestprefix` and `system` parameters:

```
>>> import bitmath
>>> print bitmath.getsize('./bitmath/__init__.py')
33.3583984375 KiB
```

Let's say we want to see the results in bytes. We can do this by setting `bestprefix` to `False`:

```
>>> import bitmath
>>> print bitmath.getsize('./bitmath/__init__.py', bestprefix=False)
34159.0 Byte
```

Recall, the default for representation is with the best human-readable prefix. We can control the prefix system used by setting `system` to either *bitmath.NIST* (the default) or *bitmath.SI*:

```
1 >>> print bitmath.getsize('./bitmath/__init__.py')
2 33.3583984375 KiB
3 >>> print bitmath.getsize('./bitmath/__init__.py', system=bitmath.NIST)
4 33.3583984375 KiB
5 >>> print bitmath.getsize('./bitmath/__init__.py', system=bitmath.SI)
6 34.159 kB
```

We can see in lines **1** → **4** that the same result is returned when `system` is not set and when `system` is set to *bitmath.NIST* (the default).

New in version 1.0.7.

bitmath.listdir()

bitmath.**listdir**(*search_base*[, *followlinks=False*[, *filter='*'*[, *relpath=False*[, *bestprefix=False*[, *system=NIST*]]]]])

This is a generator which recurses a directory tree yielding 2-tuples of:

- The absolute/relative path to a discovered file
- A bitmath instance representing the *apparent size* of the file

Parameters

- **search_base** (*string*) – The directory to begin walking down
- **followlinks** (*bool*) – **Default:** `False`, do not follow links. Whether or not to follow symbolic links to directories. Setting to `True` enables directory link following
- **filter** (*string*) – **Default:** `*` (everything). A glob to filter results with. See fnmatch for more details about *globs*
- **relpath** (*bool*) – **Default:** `False`, returns the fully qualified to each discovered file. `True` to return the relative path from the present working directory to the discovered file. If `relpath` is `False`, then *bitmath.listdir()* internally calls `os.path.realpath()` to normalize path references
- **bestprefix** (*bool*) – **Default:** `False`, returns `bitmath.Byte` instances. Set to `True` to return the best human-readable prefix unit for representation
- **system** (One of *bitmath.NIST* or *bitmath.SI*) – **Default:** *bitmath.NIST*. Set a prefix preferred unit system. Requires `bestprefix` is `True`

Note:

- This function does **not** return tuples for directory entities. Including directories in results is scheduled for introduction in the upcoming 1.1.0 release.
- Symlinks to **files** are followed automatically

When interpreting the results from this function it is *crucial* to understand exactly which items are being taken into account, what decisions were made to select those items, and how their sizes are measured.

Results from this function may seem invalid when directly compared to the results from common command line utilities, such as `du`, or `tree`.

Let's pretend we have a directory structure like the following:

```
some_files/
-- deeper_files/
|   -- second_file
-- first_file
```

Where `some_files/` is a directory, and so is `some_files/deeper_files/`. There are two regular files in this tree:

- `somefiles/first_file` - 1337 Bytes
- `some_files/deeper_files/second_file` - 13370 Bytes

The **total** size of the files in this tree is **1337 + 13370 = 14707** bytes.

Let's call *`bitmath.listdir()`* on the `some_files/` directory and see what the results look like. First we'll use all the default parameters, then we'll set `relpath` to `True`:

```
1  >>> import bitmath
2  >>> for f in bitmath.listdir('./some_files'):
3  ...     print f
4  ...
5  ('/tmp/tmp.P5lqtyqwPh/some_files/first_file', Byte(1337.0))
6  ('/tmp/tmp.P5lqtyqwPh/some_files/deeper_files/second_file', Byte(13370.0))
7  >>> for f in bitmath.listdir('./some_files', relpath=True):
8  ...     print f
9  ...
10 ('some_files/first_file', Byte(1337.0))
11 ('some_files/deeper_files/second_file', Byte(13370.0))
```

On lines **5** and **6** the results print the full path, whereas on lines **10** and **11** the path is relative to the present working directory.

Let's play with the `filter` parameter now. Let's say we only want to include results for files whose name begins with "second":

```
>>> for f in bitmath.listdir('./some_files', filter='second*'):
...     print f
...
('/tmp/tmp.P5lqtyqwPh/some_files/deeper_files/second_file', Byte(13370.0))
```

If we wish to avoid having to write for-loops, we can collect the results into a list rather simply:

```
>>> files = list(bitmath.listdir('./some_files'))
>>> print files
[('/tmp/tmp.P5lqtyqwPh/some_files/first_file', Byte(1337.0)), ('/tmp/tmp.P5lqtyqwPh/some_files/d
```

Here's a more advanced example where we will sum the size of all the returned results and then play around with the possible formatting. Recall that a bitmath instance representing the size of the discovered file is the second item in each returned tuple.

```
>>> discovered_files = [f[1] for f in bitmath.listdir('./some_files')]
>>> print discovered_files
[Byte(1337.0), Byte(13370.0)]
>>> print reduce(lambda x,y: x+y, discovered_files)
14707.0 Byte
>>> print reduce(lambda x,y: x+y, discovered_files).best_prefix()
14.3623046875 KiB
>>> print reduce(lambda x,y: x+y, discovered_files).best_prefix().format("{value:.3f} {unit}")
14.362 KiB
```

New in version 1.0.7.

bitmath.parse_string()

bitmath.**parse_string**(*str_repr*)

New in version 1.1.0.

Parse a string representing a unit into a proper bitmath object. All non-string inputs are rejected and will raise a `ValueError`. Strings without units are also rejected. See the examples below for additional clarity.

Parameters **str_repr** (*string*) – The string to parse. May contain whitespace between the value and the unit.

Returns A bitmath object representing `str_repr`

Raises **ValueError** – if `str_repr` can not be parsed

A simple usage example:

```
>>> import bitmath
>>> a_dvd = bitmath.parse_string("4.7 GiB")
>>> print type(a_dvd)
<class 'bitmath.GiB'>
>>> print a_dvd
4.7 GiB
```

Caution: Caution is advised if you are reading values from an unverified external source, such as output from a shell command or a generated file. Many applications (even `/usr/bin/ls`) still do not produce file size strings with valid (or even correct) prefix units unless specially configured to do so. See *bitmath.parse_string_unsafe()* as an alternative.

To protect your application from unexpected runtime errors it is recommended that calls to *bitmath.parse_string()* are wrapped in a `try` statement:

```
>>> import bitmath
>>> try:
...     a_dvd = bitmath.parse_string("4.7 G")
... except ValueError:
...    print "Error while parsing string into bitmath object"
...
Error while parsing string into bitmath object
```

Here we can see some more examples of invalid input, as well as two acceptable inputs:

```
>>> import bitmath
>>> sizes = [ 1337, 1337.7, "1337", "1337.7", "1337 B", "1337B" ]
>>> for size in sizes:
...     try:
```

```
...         print "Parsed size into %s" % bitmath.parse_string(size).best_prefix()
...     except ValueError:
...         print "Could not parse input: %s" % size
...
Could not parse input: 1337
Could not parse input: 1337.7
Could not parse input: 1337
Could not parse input: 1337.7
Parsed size into 1.3056640625 KiB
Parsed size into 1.3056640625 KiB
```

Changed in version 1.2.4: Added support for parsing *octet* units via issue #53 - parse french units. The usage of "octet" is still common in some RFCs, as well as France, French Canada and Romania. See also, a table of the octet units and their values on Wikipedia.

Here are some simple examples of parsing *octet* based units:

```
1 >>> import bitmath
2 >>> a_mebibyte = bitmath.parse_string("1 MiB")
3 >>> a_mebioctet = bitmath.parse_string("1 Mio")
4 >>> print a_mebibyte, a_mebioctet
5 1.0 MiB 1.0 MiB
6 >>> print bitmath.parse_string("1Po")
7 1.0 PB
8 >>> print bitmath.parse_string("1337 Eio")
9 1337.0 EiB
```

Notice how on lines **4** and **5** that the variable `a_mebibyte` from the input `1 MiB` is exactly equivalent to `a_mebioctet` from the different input `1 Mio`. This is because after *bitmath* parses the octet units the results are normalized into their **standard** NIST/SI equivalents automatically.

Note: If your input isn't compatible with *bitmath.parse_string()* you can try using *bitmath.parse_string_unsafe()* instead. *bitmath.parse_string_unsafe()* is more forgiving with input. Please read the documentation carefully so you understand the risks you assume using the `unsafe` parser.

---

bitmath.parse_string_unsafe()

`bitmath.`**`parse_string_unsafe`**(*repr*[, *system=bitmath.SI*])

New in version 1.3.1.

Parse a string or number into a proper bitmath object. This is the less strict version of the *bitmath.parse_string()* function. While *bitmath.parse_string()* only accepts SI and NIST defined unit prefixes, *bitmath.parse_string_unsafe()* accepts *non-standard* units such as those often displayed in command-line output. Examples following the description.

Parameters

- **`repr`** – The value to parse. May contain whitespace between the value and the unit.
- **`system`** – *bitmath.parse_string_unsafe()* defaults to parsing units as `SI` (base-10) units. Set the `system` parameter to *bitmath.NIST* if you know your input is in `NIST` (base-2) format.

Returns A bitmath object representing `repr`

Raises **`ValueError`** – if `repr` can not be parsed

Use of this function comes with several caveats:

- All inputs are assumed to be byte-based (as opposed to bit based)
- Numerical inputs (those without any units) are assumed to be a number of bytes
- Inputs with single letter units (`k`, `M`, `G`, etc) are assumed to be SI units (base-10). See the `system` parameter description **above** to change this behavior
- Inputs with an `i` character following the leading letter (`Ki`, `Mi`, `Gi`) are assumed to be NIST units (base-2)
- Capitalization does not matter

What exactly are these *non-standard* units? Generally speaking non-standard units will not include enough information to be able to identify exactly which unit system is being used. This is caused by mis-capitalized characters (capital `k`'s for SI *kilo* units when they should be lower case), or omitted Byte or Bit suffixes. You can find examples of non-standard units in many common command line functions or parameters. For example:

- The `ls` command will print out single-letter units when given the `-h` option flag
- Running `qemu-img info virtualdisk.img` will also report with single letter units
- The `df` command also uses single-letter units
- Kubernetes will display items like *memory limits* using two letter NIST units (ex: `memory:  2370Mi`)

Given those considerations, understanding exactly what values you are feeding into this function is crucial to getting accurate results. You can control the output of some commands with various option flags. For example, you could ensure the GNU `ls` and `df` commands print with SI values by providing the `--si` option flag. By default those commands will print out using NIST (base-2) values.

In this example let's pretend we're parsing the output of running `df -H / /boot /home` on our filesystems. Assume the output is saved into a file called `/tmp/df-output.txt` and looks like this:

```
Filesystem                                      Size  Used Avail Use% Mounted on
/dev/mapper/luks-ca8d5493-72bb-4691-afe1   107G   64G   38G  63% /
/dev/sda1                                       500M  391M   78M  84% /boot
/dev/mapper/vg_deepfryer-lv_home           129G  118G  4.7G  97% /home
```

Now let's read this file, parse the `Used` column, and then print out the space used (line **7**):

```
1  >>> with open('/tmp/df-output.txt', 'r') as fp:
2  ...     # Skip parsing the 'df' header column
3  ...     _ = fp.readline()
4  ...     for line in fp.readlines():
5  ...         cols = line.split()[0:4]
6  ...         print """Filesystem: %s
7  ... - Used: %s""" % (cols[0], bitmath.parse_string_unsafe(cols[1]))
8  Filesystem: /dev/mapper/luks-ca8d5493-72bb-4691-afe1
9  - Used: 107.0 GB
10 Filesystem: /dev/sda1
11 - Used: 500.0 MB
12 Filesystem: /dev/mapper/vg_deepfryer-lv_home
13 - Used: 129.0 GB
```

If we had ran the `df` command with the `-h` option (instead of `-H`) we will get base-2 (NIST) output. That would look like this:

```
Filesystem                                      Size  Used Avail Use% Mounted on
/dev/mapper/luks-ca8d5493-72bb-4691-afe1   100G   59G   36G  63% /
/dev/sda1                                       477M  373M   75M  84% /boot
/dev/mapper/vg_deepfryer-lv_home           120G  110G  4.4G  97% /home
```

Because we switch from `SI` output to `NIST` output the values displayed are slightly different. **However** they still print using the same prefix unit, `G`. We can tell *`bitmath.parse_string_unsafe()`* that the input is `NIST` (base-2) by giving `bitmath.NIST` to the `system` parameter like this (line **8**):

```
1  >>> with open('/tmp/df-output.txt', 'r') as fp:
2  ...     # Skip parsing the 'df' header column
3  ...     _ = fp.readline()
4  ...     for line in fp.readlines():
5  ...         cols = line.split()[0:4]
6  ...         print """Filesystem: %s
7  ... - Used: %s""" % (cols[0],
8  ...                  bitmath.parse_string_unsafe(cols[1], \
9  ...                      system=bitmath.NIST))
10 Filesystem: /dev/mapper/luks-ca8d5493-72bb-4691-afe1
11 - Used: 100.0 GiB
12 Filesystem: /dev/sda1
13 - Used: 477.0 MiB
14 Filesystem: /dev/mapper/vg_deepfryer-lv_home
15 - Used: 120.0 GiB
```

The results printed use the proper NIST prefix unit syntax now: Capital **G** followed by a lower-case **i** ending with a capital **B**, `GiB`.

bitmath.query_device_capacity()

`bitmath.`**`query_device_capacity`**(*device_fd*)

Create *`bitmath.Byte`* instances representing the capacity of a block device.

Parameters **`device_fd`** (*`file`*) – An open file handle (`handle = open('/dev/sda')`) of the target device.

Returns A *`bitmath.Byte`* equal to the size of `device_fd`

Raises

- **`ValueError`** – if file descriptor `device_fd` is not of a device type
- **`IOError`** –
  - `IOError[13]` - If the effective **uid** of this process does not have access to issue raw commands to block devices. I.e., this process does not have super-user rights
  - `IOError[2]` - If the device `device_fd` points to does not exist

Important: Superuser (root/admin) privileges are required to allow *`bitmath.query_device_capacity()`* to make the low-level system calls to read a devices capacity. Use of this function on a device the user does not have access to will result in run-time errors.

Examples of supported devices include:

•Standard Hard Drives/External Drives

•Filesystem Partitions

•Loop Devices

•LVM Logical Volumes

•Encrypted LUKS Volumes

•iSCSI Devices

Here's an example using the `with` context manager to open a device and print its capacity with the best-human readable prefix (line **3**):

```
1 >>> import bitmath
2 >>> with open("/dev/sda") as device:
3 ...     size = bitmath.query_device_capacity(device).best_prefix()
4 ...     print "Device %s capacity: %s (%s Bytes)" % (device.name, size, size_bytes)
5 Device /dev/sda capacity: 238.474937439 GiB (2.56060514304e+11 Bytes)
```

Important: Platform Notice: *bitmath.query_device_capacity()* is only verified to work on **Linux** and **Mac OS X** platforms. To file a bug report, please follow the instructions in the *contributing section*.

New in version 1.2.4.

2.1.2 Context Managers

This section describes all of the context managers provided by the bitmath class.

Note: For a bit of background, a *context manager* (specifically, the `with` statement) is a feature of the Python language which is commonly used to:

- Decorate, or *wrap*, an arbitrary block of code. I.e., effect a certain condition onto a specific body of code
- Automatically *open* and *close* an object which is used in a specific context. I.e., handle set-up and tear-down of objects in the place they are used.

See also:

PEP 343 *The "with" Statement*

PEP 318 *Decorators for Functions and Methods*

bitmath.format()

bitmath.**format**([*fmt_str=None*[, *plural=False*[, *bestprefix=False*]]])

The *bitmath.format()* context manager allows you to specify the string representation of all bitmath instances within a specific block of code.

This is effectively equivalent to applying the *format()* method to an entire region of code.

Parameters

- **fmt_str** (*str*) – a formatting mini-language compat formatting string. See the *instance attributes* for a list of available items.
- **plural** (*bool*) – `True` enables printing instances with trailing **s**'s if they're plural. `False` (default) prints them as singular (no trailing 's')
- **bestprefix** (*bool*) – `True` enables printing instances in their best human-readable representation. `False`, the default, prints instances using their current prefix unit.

Note: The `bestprefix` parameter is not yet implemented!

Let's look at an example of toggling pluralization on and off. First we'll look over a demonstration script (below), and then we'll review the output.

```
1  import bitmath
2
3  a_single_bit = bitmath.Bit(1)
4  technically_plural_bytes = bitmath.Byte(0)
5  always_plural_kbs = bitmath.kb(42)
6
7  formatting_args = {
8      'not_plural': a_single_bit,
9      'technically_plural': technically_plural_bytes,
10     'always_plural': always_plural_kbs
11 }
12
13 print """None of the following will be pluralized, because that feature is turned off
14 """
15
16 test_string = """   One unit of 'Bit': {not_plural}
17
18    0 of a unit is typically said pluralized in US English: {technically_plural}
19
20    several items of a unit will always be pluralized in normal US English
21    speech: {always_plural}"""
22
23 print test_string.format(**formatting_args)
24
25 print """
26 ------------------------------------------------------------------------
27 """
28
29 print """Now, we'll use the bitmath.format() context manager
30 to print the same test string, but with pluralization enabled.
31 """
32
33 with bitmath.format(plural=True):
34     print test_string.format(**formatting_args)
```

The context manager is demonstrated in lines **33** → **34**. In these lines we use the *`bitmath.format()`* context manager, setting `plural` to `True`, to print the original string again. By doing this we have enabled pluralized string representations (where appropriate). Running this script would have the following output:

```
None of the following will be pluralized, because that feature is turned off

   One unit of 'Bit': 1.0 Bit

   0 of a unit is typically said pluralized in US English: 0.0 Byte

   several items of a unit will always be pluralized in normal US English
   speech: 42.0 kb

------------------------------------------------------------------------

Now, we'll use the bitmath.format() context manager
to print the same test string, but with pluralization enabled.

   One unit of 'Bit': 1.0 Bit

   0 of a unit is typically said pluralized in US English: 0.0 Bytes
```

```
    several items of a unit will always be pluralized in normal US English
    speech: 42.0 kbs
```

Here's a shorter example, where we'll:

- Print a string containing bitmath instances using the default formatting (lines **2** → **3**)
- Use the context manager to print the instances in scientific notation (lines **4** → **7**)
- Print the string one last time to demonstrate how the formatting automatically returns to the default format (lines **8** → **9**)

```
1 >>> import bitmath
2 >>> print "Some instances: %s, %s" % (bitmath.KiB(1 / 3.0), bitmath.Bit(512))
3 Some instances: 0.333333333333 KiB, 512.0 Bit
4 >>> with bitmath.format("{value:e}-{unit}"):
5 ...     print "Some instances: %s, %s" % (bitmath.KiB(1 / 3.0), bitmath.Bit(512))
6 ...
7 Some instances: 3.333333e-01-KiB, 5.120000e+02-Bit
8 >>> print "Some instances: %s, %s" % (bitmath.KiB(1 / 3.0), bitmath.Bit(512))
9 Some instances: 0.333333333333 KiB, 512.0 Bit
```

New in version 1.0.8.

2.1.3 Module Variables

This section describes the module-level variables. Some of which are constants and are used for reference. Some of which effect output or behavior.

Changed in version 1.0.7: The formatting strings were not available for manupulate/inspection in earlier versions

New in version 1.1.1: Prior to this version *ALL_UNIT_TYPES* was not defined

---

Note: Modifying these variables will change the default representation indefinitely. Use the *bitmath.format()* context manager to limit changes to a specific block of code.

---

bitmath.**format_string**

This is the default string representation of all bitmath instances. The default value is `{value} {unit}` which, when evaluated, formats an instance as a floating point number with at least one digit of precision, followed by a character of whitespace, followed by the prefix unit of the instance.

For example, given bitmath instances representing the following values: **1337 MiB**, **0.1234567 kb**, and **0 B**, their printed output would look like the following:

```
>>> from bitmath import *
>>> print MiB(1337), kb(0.1234567), Byte(0)
1337.0 MiB 0.1234567 kb 0.0 Byte
```

We can make these instances print however we want to. Let's wrap each one in square brackets (`[`, `]`), replace the separating space character with a hyphen (`-`), and limit the precision to just 2 digits:

```
>>> import bitmath
>>> bitmath.format_string = "[{value:.2f}-{unit}]"
>>> print bitmath.MiB(1337), bitmath.kb(0.1234567), bitmath.Byte(0)
[1337.00-MiB] [0.12-kb] [0.00-Byte]
```

bitmath.**format_plural**
A boolean which controls the pluralization of instances in string representation. The default is `False`.

If we wanted to enable pluralization we could set the *format_plural* variable to `True`. First, let's look at some output using the default singular formatting.

```
>>> import bitmath
>>> print bitmath.MiB(1337)
1337.0 MiB
```

And now we'll enable pluralization (line **2**):

```
1 >>> import bitmath
2 >>> bitmath.format_plural = True
3 >>> print bitmath.MiB(1337)
4 1337.0 MiBs
5 >>> bitmath.format_plural = False
6 >>> print bitmath.MiB(1337)
7 1337.0 MiB
```

On line **5** we disable pluralization again and then see that the output has no trailing "s" character.

bitmath.**NIST**
Constant used as an argument to some functions to specify the **NIST** system.

bitmath.**SI**
Constant used as an argument to some functions to specify the **SI** system.

bitmath.**SI_PREFIXES**
An array of all of the SI unit prefixes (e.g., `k`, `M`, or `E`)

bitmath.**SI_STEPS**

```
SI_STEPS = {
    'Bit': 1 / 8.0,
    'Byte': 1,
    'k': 1000,
    'M': 1000000,
    'G': 1000000000,
    'T': 1000000000000,
    'P': 1000000000000000,
    'E': 1000000000000000000
}
```

bitmath.**NIST_PREFIXES**
An array of all of the NIST unit prefixes (e.g., `Ki`, `Mi`, or `Ei`)

bitmath.**NIST_STEPS**

```
NIST_STEPS = {
    'Bit': 1 / 8.0,
    'Byte': 1,
    'Ki': 1024,
    'Mi': 1048576,
    'Gi': 1073741824,
    'Ti': 1099511627776,
    'Pi': 1125899906842624,
    'Ei': 1152921504606846976
}
```

bitmath.**ALL_UNIT_TYPES**
An array of all combinations of known valid prefix units mixed with both bit and byte suffixes.

```
ALL_UNIT_TYPES = ['b', 'B', 'kb', 'kB', 'Mb', 'MB', 'Gb', 'GB',
   'Tb', 'TB', 'Pb', 'PB', 'Eb', 'EB', 'Kib', 'KiB', 'Mib',
   'MiB', 'Gib', 'GiB', 'Tib', 'TiB', 'Pib', 'PiB', 'Eib',
   'EiB']
```

2.1.4 3rd Party Module Integrations

This section describes the various ways in which *bitmath* can be integrated with other 3rd pary modules.

To see a full demo of the argparse and progressbar integrations, as well as a comprehensive demonstrations of the full capabilities of the bitmath library, see *Creating Download Progress Bars* in the *Real Life Examples* section.

argparse

New in version 1.2.0.

The argparse module (part of stdlib) is used to parse command line arguments. By default, parsed options and arguments are turned into strings. However, one useful feature argparse provides is the ability to specify what datatype any given argument or option should be interpreted as.

bitmath.integrations.**BitmathType**(*bmstring*)
The *BitmathType()* factory creates objects that can be passed to the type argument of ArgumentParser.add_argument(). Arguments that have *BitmathType()* objects as their type will automatically parse the command line argument into a matching *bitmath object*.

Parameters **bmstring** (*str*) – The command-line option to parse into a bitmath object

Returns A bitmath object representing bmstring

Raises

- **ValueError** – on any input that *bitmath.parse_string()* already rejects
- **ValueError** – on **unquoted inputs** with whitespace separating the value from the unit (e.g., --some-option 10 MiB is bad, but --some-option '10 MiB' is good)

Let's take a look at a more in-depth example.

A feature found in many command-line utilities is the ability to specify some kind of file size using a string which roughly describes some kind of parameter. For example, let's look at the **du** (disk usage) command. Invoking it as du -B allows one to specify a desired block-size scaling factor in printed results.

Let's say we wanted to implement a similar mechanism in an application of our own. Except, instead of abbreviating down to ambiguous capital letters, we accept scaling factors as *properly written values* with associated units. Such as **10 MiB**, or **1 MB**.

To accomplish this, we'll use argparse to create an argument parser and add one option to it, --block-size. This option will have a type of *BitmathType()* set.

```
>>> import argparse, bitmath
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--block-size', type=bitmath.BitmathType)
>>> args = "--block-size 1MiB"
>>> results = parser.parse_args(args.split())
>>> print type(results.block_size)
<class 'bitmath.MiB'>
```

On line **3** we add the `--block-size` option to the parser, explicitly defining it's type as *`BitmathType()`*. In lines **6** and **7** when we parse the provided arguments we find that `argparse` has automatically created a bitmath object for us.

If an invalid scaling factor is provided by the user, such as one which does not represent a recognizable unit, the bitmath library will automatically detect this for us and signal to the argument parser that an error has occurred.

progressbar

New in version 1.2.1.

The progressbar module is typically used to display the progress of a long running task, such as a file transfer operation. The module provides widgets for custom formatting how exactly the 'progress' is displayed. Some examples include: overall percentage complete, estimated time until completion, and an ASCII progress bar which fills as the operation continues.

While `progressbar` already includes a widget suitable for displaying file transfer rates, this widget does not support customizing its presentation, and is limited to only prefix units from the SI system.

class `bitmath.integrations.`**`BitmathFileTransferSpeed`**(*[system=bitmath.NIST[, format="{value:.2f} {unit}/s"]]*)

The *`BitmathFileTransferSpeed`* class is a more functional replacement for the upstream FileTransferSpeed widget.

While both widgets are able to calculate average transfer rates over a period of time, the *`BitmathFileTransferSpeed`* widget adds new support for NIST prefix units (the upstream widget only supports SI prefix units).

In addition to NIST unit support, *`BitmathFileTransferSpeed`* enables the user to have **full control** over the look and feel of the displayed rates.

Parameters

- **`system`** (One of *`bitmath.NIST`* or *`bitmath.SI`*) – **Default:** *`bitmath.NIST`*. The preferred system of units for the printed rate.
- **`format`** (*`string`*) – a formatting mini-language compat formatting string. **Default** `{value:.2f} {unit}/s` (e.g., `13.37 GiB/s`)

Note: See *instance attributes* for a list of available formatting items. See the section on *formatting bitmath instances* for more information on this topic.

Use *`BitmathFileTransferSpeed`* exactly like the upstream `FileTransferSpeed` widget (example copied and modified from the progressbar project page):

```
>>> from progressbar import ProgressBar, Percentage, Bar, ETA, RotatingMarker
>>> from bitmath.integrations import BitmathFileTransferSpeed
>>> widgets = ['Something: ', Percentage(), ' ', Bar(marker=RotatingMarker()),
...            ' ', ETA(), ' ', BitmathFileTransferSpeed()]
>>> pbar = ProgressBar(widgets=widgets, maxval=10000000).start()
>>> for i in range(1000000):
...     # do something
...     pbar.update(10*i+1)
>>> pbar.finish()
```

If this was ran from a script we would see output similar to the following:

```
Something: 100% |||||||||||||||||||||||||||||||||||| Time: 0:00:01 9.27 MiB/s
```

If we wanted behavior identical to `FileTransferSpeed` we would set the `system` parameter to *bitmath.SI* (line **5** below):

```
1 >>> import bitmath
2 >>> # ...
3 >>> widgets = ['Something: ', Percentage(), ' ', Bar(marker=RotatingMarker()),
4 ...            ' ', ETA(), ' ',
5 ...            BitmathFileTransferSpeed(system=bitmath.SI)]
6 >>> pbar = ProgressBar(widgets=widgets, maxval=10000000).start()
7 >>> # ...
```

If this was ran from a script we would see output similar to the following:

```
Something: 100% |||||||||||||||||||||||||||||||||||| Time: 0:00:01 9.80 MB/s
```

Note how the only difference is in the displayed unit. The former example produced a rate with a unit of `MiB` (a NIST unit) whereas the latter examples unit is `MB` (an SI unit).

As noted previously, *BitmathFileTransferSpeed* allows for full control over the formatting of the calculated rate of transfer.

For example, if we wished to see the rate printed using more verbose language and plauralized units, we could do exactly that by constructing our widget in the following way:

```
BitmathFileTransferSpeed(format="{value:.2f} {unit_plural} per second")
```

And if this were run from a script like the previous examples:

```
Something: 100% |||||||||||||||||||||||||||||||||||| Time: 0:00:01 9.41 MiBs per second
```

2.2 The `bitmath` command-line Tool

bitmath includes a CLI utility for easily converting units in a shell. For reference, there is also a manpage included, *bitmath (1)*.

2.2.1 Synopsis

```
bitmath [--from-stdin] [-f IN_UNIT] [-t OUT_UNIT] VALUE ...
```

2.2.2 Options

-f <IN_UNIT>
: Specify the input unit to convert from. Defaults to *bitmath.Byte*.

-t <OUT_UNIT>
: Specify the output unit to convert to. Defaults to the *best human-readable* prefix unit.

--from-stdin
: Reads number from stdin rather than as a CLI argument.

VALUE
: The value to convert.

2.2.3 Examples

Convert `1024` into the best human-readable unit. Without specifying any `from` or `to` values this examples defaults to treating the input value as a *`bitmath.Byte`*:

```
$ bitmath 1024
1.0 KiB
```

Convert 1024 KiB into kBs:

```
$ bitmath -f KiB -t kb 1024
8388.608 kb
```

Convert 1073741824 bytes into the best human-readable unit:

```
$ bitmath -f Byte 1073741824
1.0 GiB
```

Use the **`stat`** command to print the size of `bitmath/__init__.py` in bytes, pipe the output into the **`bitmath`** command, and print the result in MBs:

```
$ stat -c '%s' bitmath/__init__.py | bitmath --from-stdin -t MB
0.038374 MB
```

Convert several values at once from Bytes (the default behavior) into MBs:

```
$ bitmath -t MB 1234567 9876543 1337 42
1.234567 MB
9.876543 MB
0.001337 MB
4.2e-05 MB
```

2.3 Classes

2.3.1 Available Classes

There are two **fundamental** classes available, the *`Bit`* and the *`Byte`*.

There are **24** other classes available, representing all the prefix units from **k** through **e** (*kilo/kibi* through *exa/exbi*).

Classes with **'i'** in their names are **NIST** type classes. They were defined by the National Institute of Standards and Technology (NIST) as the 'Binary Prefix Units'. They are defined by increasing powers of 2.

Classes without the **'i'** character are **SI** type classes. Though not formally defined by any standards organization, they follow the International System of Units (SI) pattern (commonly used to abbreviate base 10 values). You may hear these referred to as the "Decimal" or "SI" prefixes.

Classes ending with *lower-case* **'b'** characters are **bit based**. Classes ending with upper-case **'B'** characters are **byte based**. Class inheritance is shown below in parentheses to make this more apparent:

NIST	SI
`Eib(Bit)`	`Eb(Bit)`
`EiB(Byte)`	`EB(Byte)`
`Gib(Bit)`	`Gb(Bit)`
`GiB(Byte)`	`GB(Byte)`
`Kib(Bit)`	`kb(Bit)`
`KiB(Byte)`	`kB(Byte)`
`Mib(Bit)`	`Mb(Bit)`
`MiB(Byte)`	`MB(Byte)`
`Pib(Bit)`	`Pb(Bit)`
`PiB(Byte)`	`PB(Byte)`
`Tib(Bit)`	`Tb(Bit)`
`TiB(Byte)`	`TB(Byte)`

Note: As per SI definition, the `kB` and `kb` classes begins with a *lower-case* **k** character.

The majority of the functionality of bitmath object comes from their rich implementation of standard Python operations. You can use bitmath objects in **almost all** of the places you would normally use an integer or a float. See the *Table of Supported Operations* and *Appendix: Rules for Math* for more details.

2.3.2 Initializing

class `bitmath.`**`Bit`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Byte`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`EB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Eb`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`EiB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Eib`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`GB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Gb`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`GiB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Gib`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`kB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`kb`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`KiB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Kib`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`MB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Mb`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`MiB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`Mib`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class `bitmath.`**`PB`**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Pb**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**PiB**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Pib**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**TB**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Tb**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**TiB**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Tib**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**YB**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Yb**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**ZB**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Zb**([*value=0*[, *bytes=None*[, *bits=None*]]])

class bitmath.**Bitmath**([*value=0*[, *bytes=None*[, *bits=None*]]])

The `value`, `bytes`, and `bits` parameters are **mutually exclusive**. That is to say, you cannot instantiate a bitmath class using more than **one** of the parameters. Omitting any keyword argument defaults to behaving as if `value` was provided.

Parameters

- **value** (*int*) – **Default: 0**. The value of the instance in *prefix units*. For example, if we were instantiating a `bitmath.KiB` object to represent 13.37 KiB, the `value` parameter would be **13.37**. For instance, `k = bitmath.KiB(13.37)`.
- **bytes** (*int*) – The value of the instance as measured in bytes.
- **bits** (*int*) – The value of the instance as measured in bits.

Raises **ValueError** – if more than one parameter is provided.

The following code block demonstrates the 4 acceptable ways to instantiate a bitmath class.

```
>>> import bitmath

# Omitting all keyword arguments defaults to 'value' behavior.
>>> a = bitmath.KiB(1)

# This is equivalent to the previous statement
>>> b = bitmath.KiB(value=1)

# We can also specify the initial value in bytes.
# Recall, 1KiB = 1024 bytes
>>> c = bitmath.KiB(bytes=1024)

# Finally, we can specify exact number of bits in the
# instance. Recall, 1024B = 8192b
>>> d = bitmath.KiB(bits=8192)

>>> a == b == c == d
True
```

2.3.3 Class Methods

Class Method: from_other()

bitmath **class objects** have one public class method, `BitMathClass.from_other()` which provides an alternative way to initialize a bitmath class.

This method may be called on bitmath class objects directly. That is to say: you do not need to call this method on an instance of a bitmath class, however that is a valid use case.

classmethod `Byte.`**`from_other`**(*item*)

Instantiate any `BitMathClass` using another instance as reference for it's initial value.

The `from_other()` class method has one required parameter: an instance of a bitmath class.

Parameters **`item`** (*`BitMathInstance`*) – An instance of a bitmath class.

Returns a bitmath instance of type `BitMathClass` equivalent in value to `item`

Return type BitMathClass

Raises **`TypeError`** – if `item` is not a valid *bitmath class*

In pure Python, this could also be written as:

```
>>> import bitmath

>>> a_mebibyte = bitmath.MiB(1)

>>> a_mebibyte_sized_kibibyte = bitmath.KiB(bytes=a_mebibyte.bytes)

>>> a_mebibyte == a_mebibyte_sized_kibibyte
True

>>> print a_mebibyte, a_mebibyte_sized_kibibyte
1.0 MiB 1024.0 KiB
```

Or, using the `BitMathClass.from_other()` class method:

```
>>> a_mebibyte = bitmath.MiB(1)

>>> a_big_kibibyte = bitmath.KiB.from_other(a_mebibyte)

>>> a_mebibyte == a_big_kibibyte
True

>>> print a_mebibyte, a_big_kibibyte
1.0 MiB 1024.0 KiB
```

2.4 Instances

2.4.1 Instance Attributes

bitmath objects have several instance attributes:

`BitMathInstance.`**`base`**
The mathematical base of the unit of the instance (this will be **2** or **10**)

```
>>> b = bitmath.Byte(1337)
>>> print b.base
2
```

`BitMathInstance.`**`binary`**
The Python binary representation of the instance's value (in bits)

```
>>> b = bitmath.Byte(1337)
>>> print b.binary
0b10100111001000
```

`BitMathInstance.`**`bin`**
This is an alias for `binary`

`BitMathInstance.`**`bits`**
The number of bits in the object

```
>>> b = bitmath.Byte(1337)
>>> print b.bits
10696.0
```

`BitMathInstance.`**`bytes`**
The number of bytes in the object

```
>>> b = bitmath.Byte(1337)
>>> print b.bytes
1337
```

`BitMathInstance.`**`power`**
The mathematical power the `base` of the unit of the instance is raised to

```
>>> b = bitmath.Byte(1337)
>>> print b.power
0
```

`BitMathInstance.`**`system`**
The system of units used to measure this instance (`NIST` or `SI`)

```
>>> b = bitmath.Byte(1337)
>>> print b.system
NIST
```

`BitMathInstance.`**`value`**
The value of the instance in *prefix* units[1]

```
>>> b = bitmath.Byte(1337)
>>> print b.value
1337.0
```

`BitMathInstance.`**`unit`**
The string representation of this prefix unit (such as `MiB` or `kb`)

```
>>> b = bitmath.Byte(1337)
>>> print b.unit
Byte
```

`BitMathInstance.`**`unit_plural`**
The pluralized string representation of this prefix unit.

```
>>> b = bitmath.Byte(1337)
>>> print b.unit_plural
Bytes
```

`BitMathInstance.`**`unit_singular`**
The singular string representation of this prefix unit (such as `MiB` or `kb`)

```
>>> b = bitmath.Byte(1337)
>>> print b.unit_singular
Byte
```

Notes:

1. Given an instance `k`, where `k = KiB(1.3)`, then `k.value` is **1.3**

The following is an example of how to access some of these attributes and what you can expect their printed representation to look like:

```
>>> dvd_capacity = GB(4.7)
>>> print "Capacity in bits: %s\nbytes: %s\n" % \
              (dvd_capacity.bits, dvd_capacity.bytes)

Capacity in bits: 37600000000.0
bytes: 4700000000.0

>>> dvd_capacity.value
4.7

>>> dvd_capacity.bin
'0b100011000001001000100111100000000000'

>>> dvd_capacity.binary
'0b100011000001001000100111100000000000'
```

2.4.2 Instance Methods

bitmath objects come with a few basic methods: `to_THING()`, `format()`, and *`best_prefix()`*.

to_THING()

Like the *available classes*, there are 24 `to_THING()` methods available. `THING` is any of the bitmath classes. You can even `to_THING()` an instance into itself again:

```
>>> from bitmath import *
>>> one_mib = MiB(1)
>>> one_mib_in_kb = one_mib.to_kb()
>>> one_mib == one_mib_in_kb
True

>>> another_mib = one_mib.to_MiB()
>>> print one_mib, one_mib_in_kb, another_mib
1.0 MiB 8388.608 kb 1.0 MiB

>>> six_TB = TB(6)
>>> six_TB_in_bits = six_TB.to_Bit()
>>> print six_TB, six_TB_in_bits
6.0 TB 4.8e+13 Bit

>>> six_TB == six_TB_in_bits
True
```

best_prefix()

best_prefix([*system=None*])

Return an equivalent instance which uses the best human-readable prefix-unit to represent it.

Parameters **system** (*int*) – one of *bitmath.NIST* or *bitmath.SI*

Returns An equivalent *bitmath* instance

Return type *bitmath*

Raises **ValueError** – if an invalid unit system is given for `system`

The *best_prefix()* method returns the result of converting a bitmath instance into an equivalent instance using a prefix unit that better represents the original value. Another way to think of this is automatic discovery of the most sane, or *human readable*, unit to represent a given size. This functionality is especially important in the domain of interactive applications which need to report file sizes or transfer rates to users.

As an analog, consider you have 923,874,434¢ in your bank account. You probably wouldn't want to read your bank statement and find your balance in pennies. Most likely, your bank statement would read a balance of $9,238,744.34. In this example, the input prefix is the *cent*: `¢`. The *best prefix* for this is the *dollar*: `$`.

Let's, for example, say we are reporting a transfer rate in an interactive application. It's important to present this information in an easily consumable format. The library we're using to calculate the rate of transfer reports the rate in bytes per second from a `tx_rate()` function.

We'll use this example twice. In the first occurrence, we will print out the transfer rate in a more easily digestible format than pure bytes/second. In the second occurrence we'll take it a step further, and use the *format* method to make the output even easier to read.

```
>>> for _rate in tx_rate():
...     print "Rate: %s/second" % Bit(_rate)
...     time.sleep(1)

Rate: 100.0 Bit/sec
Rate: 24000.0 Bit/sec
```

```
Rate: 1024.0 Bit/sec
Rate: 60151.0 Bit/sec
Rate: 33.0 Bit/sec
Rate: 9999.0 Bit/sec
Rate: 9238742.0 Bit/sec
Rate: 2.09895849555e+13 Bit/sec
Rate: 934098021.0 Bit/sec
Rate: 934894.0 Bit/sec
```

And now using a custom formatting definition:

```
>>> for _rate in tx_rate():
...     print Bit(_rate).best_prefix().format("Rate: {value:.3f} {unit}/sec")
...     time.sleep(1)

Rate: 12.500 Byte/sec
Rate: 2.930 KiB/sec
Rate: 128.000 Byte/sec
Rate: 7.343 KiB/sec
Rate: 4.125 Byte/sec
Rate: 1.221 KiB/sec
Rate: 1.101 MiB/sec
Rate: 2.386 TiB/sec
Rate: 111.353 MiB/sec
Rate: 114.123 KiB/sec
```

format()

BitMathInstance.**format**(*fmt_spec*)

Return a custom-formatted string to represent this instance.

Parameters **fmt_spec** (*str*) – A valid formatting mini-language string

Returns The custom formatted representation

Return type string

bitmath instances come with a verbose built-in string representation:

```
>>> leet_bits = Bit(1337)
>>> print leet_bits
1337.0 Bit
```

However, for instances which aren't whole numbers (as in `MiB(1/3.0) == 0.333333333333 MiB`, etc), their representation can be undesirable.

The `format()` method gives you complete control over the instance's representation. All of the *instances attributes* are available to use when choosing a representation.

The following sections describe some common use cases of the `format()` method as well as provide a *brief tutorial* of the greater Python formatting meta-language.

Setting Decimal Precision

By default, bitmath instances will print to a fairly long precision for values which are not whole multiples of their prefix unit. In most use cases, simply printing out the first 2 or 3 digits of precision is acceptable.

The following examples will show us how to print out a bitmath instance in a more human readable way, by limiting the decimal precision to 2 digits.

First, for reference, the default formatting:

```
>>> ugly_number = MB(50).to_MiB() / 8.0
>>> print ugly_number
5.96046447754 MiB
```

Now, let's use the `format()` method to limit that to two digits of precision:

```
>>> print ugly_number.format("{value:.2f}{unit}")
5.96 MiB
```

By changing the **2** character, you increase or decrease the precision. Set it to **0** (`{value:.0f}`) and you have what effectively looks like an integer.

Format All the Instance Attributes

The following example prints out every instance attribute. Take note of how an attribute may be referenced multiple times.

```
1  >>> longer_format = """Formatting attributes for %s
2     ...: This instances prefix unit is {unit}, which is a {system} type unit
3     ...: The unit value is {value}
4     ...: This value can be truncated to just 1 digit of precision: {value:.1f}
5     ...: In binary this looks like: {binary}
6     ...: The prefix unit is derived from a base of {base}
7     ...: Which is raised to the power {power}
8     ...: There are {bytes} bytes in this instance
9     ...: The instance is {bits} bits large
10    ...: bytes/bits without trailing decimals: {bytes:.0f}/{bits:.0f}""" % str(ugly_number)
11
12 >>> print ugly_number.format(longer_format)
13 Formatting attributes for 5.96046447754 MiB
14 This instances prefix unit is MiB, which is a NIST type unit
15 The unit value is 5.96046447754
16 This value can be truncated to just 1 digit of precision: 6.0
17 In binary this looks like: 0b101111101011110000010000000
18 The prefix unit is derived from a base of 2
19 Which is raised to the power 20
20 There are 6250000.0 bytes in this instance
21 The instance is 50000000.0 bits large
22 bytes/bits without trailing decimals: 6250000/50000000
```

Note: On line **4** we print with 1 digit of precision, on line **16** we see the value has been rounded to **6.0**

2.4.3 Instance Properties

THING Properties

Like the *available classes*, there are 24 `THING` properties available. `THING` is any of the bitmath classes. Under the covers these properties call `to_THING`.

```
1 >>> from bitmath import *
2 >>> one_mib = MiB(1)
3 >>> one_mib == one_mib.kb
```

```
True

>>> print one_mib, one_mib.kb, one_mib.MiB
1.0 MiB 8388.608 kb 1.0 MiB

>>> six_TB = TB(6)
>>> print six_TB, six_TB.Bit
6.0 TB 4.8e+13 Bit

>>> six_TB == six_TB.Bit
True
```

2.4.4 The Formatting Mini-Language

That is all you begin printing numbers with custom precision. If you want to learn a little bit more about using the formatting mini-language, read on.

You may be asking yourself where these `{value:.2f}` and `{unit}` strings came from. These are part of the Format Specification Mini-Language which is part of the Python standard library. To be explicitly clear about what's going on here, let's break the first specifier (`{value:.2f}`) down into it's component parts:

```
{value:.2f}
   |  |||
   |  |||\---- The "f" says to format this as a floating point type
   |  ||\----- The 2 indicates we want 2 digits of precision (default is 6)
   |  |\------ The '.' character must precede the precision specifier for floats
   |  \------- The : separates the attribute name from the formatting specification
   \---------- The name of the attribute to print
```

The second specifier (`{unit}`) says to format the `unit` attribute as a string (string is the default type when no type is given).

See also:

Python String Format Cookbook Marcus Kazmierczak's *excellent* introduction to string formatting

2.5 Getting Started

In this section we will take a high-level look at the basic things you can do with bitmath. We'll include the following topics:

2.5.1 Tables of Supported Operations

The following legend describes the two operands used in the tables below.

Operand	Description
`bm`	A bitmath object is required
`num`	An integer or decimal value is required

Arithmetic

Math works mostly like you expect it to, except for a few edge-cases:

- Mixing bitmath types with Number types (the result varies per-operation)
- Operations where two bitmath types would cancel out (such as dividing two bitmath types)
- Multiplying two bitmath instances together is supported, but the results may not make much sense.

See also:

Appendix: Rules for Math For a discussion of the behavior of bitmath and number types.

Operation	Parameters	Result Type	Example
Addition	`bm1 + bm2`	`type(bm1)`	`KiB(1) + MiB(2) = 2049.0KiB`
Addition	`bm + num`	`type(num)`	`KiB(1) + 1 = 2.0`
Addition	`num + bm`	`type(num)`	`1 + KiB(1) = 2.0`
Subtraction	`bm1 - bm2`	`type(bm1)`	`KiB(1) - Byte(2048) = -1.0KiB`
Subtraction	`bm - num`	`type(num)`	`KiB(4) - 1 = 3.0`
Subtraction	`num - bm`	`type(num)`	`10 - KiB(1) = 9.0`
Multiplication	`bm1 * bm2`	`type(bm1)`	`KiB(1) * KiB(2) = 2048.0KiB`
Multiplication	`bm * num`	`type(bm)`	`KiB(2) * 3 = 6.0KiB`
Multiplication	`num * bm`	`type(bm)`	`2 * KiB(3) = 6.0KiB`
Division	`bm1 / bm2`	`type(num)`	`KiB(1) / KiB(2) = 0.5`
Division	`bm / num`	`type(bm)`	`KiB(1) / 3 = 0.3330078125KiB`
Division	`num / bm`	`type(num)`	`3 / KiB(2) = 1.5`

Bitwise Operations

See also:

Bitwise Calculator A free online calculator for checking your math

Bitwise operations are also supported. Bitwise operations work directly on the `bits` attribute of a bitmath instance, not the number you see in an instances printed representation (`value`), to maintain accuracy.

Operation	Parameters	Result Type	Example[1]
Left Shift	`bm << num`	`type(bm)`	`MiB(1) << 2 = MiB(4.0)`
Right Shift	`bm >> num`	`type(bm)`	`MiB(1) >> 2 = MiB(0.25)`
AND	`bm & num`	`type(bm)`	`MiB(13.37) & 1337 = MiB(0.000126...)`
OR	`bm | num`	`type(bm)`	`MiB(13.37) | 1337 = MiB(13.3700...)`
XOR	`bm ^ num`	`type(bm)`	`MiB(13.37) ^ 1337 = MiB(13.369...)`

1. *Give me a break here, it's not easy coming up with compelling examples for bitwise operations...*

2.5.2 Basic Math

bitmath supports all arithmetic operations

```
>>> eighty_four_mib = fourty_two_mib + fourty_two_mib_in_kib
>>> eighty_four_mib
MiB(84.0)
>>> eighty_four_mib == fourty_two_mib * 2
True
```

2.5.3 Unit Conversion

```
>>> from bitmath import *
>>> fourty_two_mib = MiB(42)
>>> fourty_two_mib_in_kib = fourty_two_mib.to_KiB()
>>> fourty_two_mib_in_kib
KiB(43008.0)

>>> fourty_two_mib
MiB(42.0)

>>> fourty_two_mib.KiB
KiB(43008.0)
```

2.5.4 Rich Comparison

Rich Comparison (as per the Python Basic Customization magic methods): <, <=, ==, !=, >, >= is fully supported:

```
>>> GB(1) < GiB(1)
True
>>> GB(1.073741824) == GiB(1)
True
>>> GB(1.073741824) <= GiB(1)
True
>>> Bit(1) == TiB(bits=1)
True
>>> kB(100) > EiB(bytes=1024)
True
>>> kB(100) >= EiB.from_other(kB(100))
True
>>> kB(100) >= EiB.from_other(kB(99))
True
>>> kB(100) >= EiB.from_other(kB(9999))
False
>>> KiB(100) != Byte(1)
True
```

2.5.5 Sorting

bitmath natively supports sorting.

Let's make a list of the size (in bytes) of all the files in the present working directory (lines **7** and **8**) and then print them out sorted by increasing magnitude (lines **13** and **14**, and **18** and **19**):

```
>>> from bitmath import *
>>> import os
>>> sizes = []
>>> for f in os.listdir('./tests/'):
```

```
...     sizes.append(KiB(os.path.getsize('./tests/' + f)))

>>> print sizes
[KiB(7337.0), KiB(1441.0), KiB(2126.0), KiB(2178.0), KiB(2326.0), KiB(4003.0), KiB(48.0), KiB(1770.0)

>>> print sorted(sizes)
[KiB(48.0), KiB(1441.0), KiB(1770.0), KiB(2126.0), KiB(2178.0), KiB(2326.0), KiB(4003.0), KiB(4190.0)

>>> human_sizes = [s.best_prefix() for s in sizes]
>>> print sorted(human_sizes)
[KiB(48.0), MiB(1.4072265625), MiB(1.728515625), MiB(2.076171875), MiB(2.126953125), MiB(2.271484375)
```

Now print them out in descending magnitude

```
>>> print sorted(human_sizes, reverse=True)
[KiB(7892.0), KiB(7337.0), KiB(4190.0), KiB(4003.0), KiB(2326.0), KiB(2178.0), KiB(2126.0), KiB(1770
```

2.6 Real Life Examples

2.6.1 Download Speeds

Let's pretend that your Internet service provider (ISP) advertises your maximum downstream as **50Mbps** (50 Mega**bits** per second)[1] and you want to know how fast that is in Mega**bytes** per second? `bitmath` can do that for you easily. We can calculate this as such:

```
>>> import bitmath
>>> downstream = bitmath.Mib(50)
>>> print downstream.to_MB()
MB(6.25)
```

This tells us that if our ISP advertises **50Mbps** we can expect to see download rates of over **6MB/sec**.

1. *Assuming your ISP follows the common industry practice of using SI (base-10) units to describe file sizes/rates*

2.6.2 Calculating how many files fit on a device

In 2001 Apple® announced the iPod™. Their headline statement boasting:

> "... iPod stores up to 1,000 CD-quality songs on its super-thin 5 GB hard drive, ..."

OK. That's pretty simple to work backwards: *capacity of disk drive* divided by *number of songs* equals the average size of a song. Which in this case is:

```
>>> song_size = GB(5) / 1000
>>> print song_size
0.005GB
```

Or, using `best_prefix`, (line **2**) to generate a more human-readable form:

```
>>> song_size = GB(5) / 1000
>>> print song_size.best_prefix()
5.0MB
```

That's great, if you have normal radio-length songs. But how many of our favorite jam-band's 15-30+ minute-long songs could we fit on this iPod? Let's pretend we did the math and the average audio file worked out to be **18.6 MiB** (19.5 MB) large.

```
>>> ipod_capacity = GB(5)
>>> bootleg_size = MB(19.5)
>>> print ipod_capacity / bootleg_size
256.41025641
```

The result on line **4** tells tells us that we could fit **256** average-quality songs on our iPod.

2.6.3 Printing Human-Readable File Sizes in Python

In a Python script or interpreter we may wish to print out file sizes in something other than bytes (which is what `os.path.getsize` returns). We can use `bitmath` to do that too:

```
>>> import os
>>> from bitmath import *
>>> these_files = os.listdir('.')
>>> for f in these_files:
...     f_size = Byte(os.path.getsize(f))
...     print "%s - %s" % (f, f_size.to_KiB())

test_basic_math.py - 3.048828125 KiB
__init__.py - 0.1181640625 KiB
test_representation.py - 0.744140625 KiB
test_to_Type_conversion.py - 2.2119140625 KiB
```

Alternatively, we could simplify things and use *bitmath.getsize()* to read the file size directly into a bitmath object:

```
>>> import os
>>> import bitmath
>>> these_files = os.listdir('.')
>>> for f in these_files:
...     print "%s - %s" % (f, bitmath.getsize(f))

test_basic_math.py - 3.048828125 KiB
__init__.py - 0.1181640625 KiB
test_representation.py - 0.744140625 KiB
test_to_Type_conversion.py - 2.2119140625 KiB
```

See also:

Instance Formatting How to print results in a *prettier* format

2.6.4 Calculating Linux BDP and TCP Window Scaling

Say we're doing some Linux Kernel TCP performance tuning. For optimum speeds we need to calculate our BDP, or Bandwidth Delay Product. For this we need to calculate certain values to set some kernel tuning parameters to. The point of this tuning is to send the most data we can during a measured round-trip-time without sending more than can be processed. To accomplish this we are resizing our kernel read/write networking/socket buffers.

We will see two ways of doing this. The tedious manual way, and the way with bitmath.

The Hard Way

Core Networking Values

- `net.core.rmem_max` - **Bytes** - Single Value - Default receive buffer size
- `net.core.wmem_max` - **Bytes** - Single Value - Default write buffer size

System-Wide Memory Limits

- `net.ipv4.tcp_mem` - **Pages** - Three Value Vector - The `max` field of the parameter is the number of **memory pages** allowed for queueing by all TCP sockets.

Per-Socket Buffers

Per-socket buffer sizes must not exceed the core networking buffer sizes.

- `net.ipv4.tcp_rmem` - **Bytes** - Three Field Vector - The `max` field sets the size of the TCP receive buffer
- `net.ipv4.tcp_wmem` - **Bytes** - Three Field Vector - As above, but for the write buffer

We would normally calculate the optimal BDP and related values following this approach:

1. Measure the latency, or round trip time (RTT, measured in milliseconds), between the host we're tuning and our target remote host
2. Measure/identify our network transfer rate
3. Calculate the BDP (multiply transfer rate by rtt)
4. Obtain our current kernel settings
5. Adjust settings as necessary

But for the sake brevity we'll be working out of an example scenario with a pre-defined RTT and transfer rate.

Scenario

- We have an average network transfer rate of **1Gb/sec** (where `Gb` is the SI unit for Gigabits, not Gibibytes: `GiB`)
- Our latency (RTT) is **0.199ms** (milliseconds)

Calculate Manually

Lets calculate the BDP now. Because the kernel parameters expect values in units of bytes and pages we'll have to convert our transfer rate of 1Gb/sec into B/s (Gigabits/second to Bytes/second):

- Convert 1Gb into an equivalent **byte** based unit

Remember 1 Byte = 8 Bits:

```
tx_rate_GB = 1/8 = 0.125
```

Our equivalent transfer rate is 0.125GB/sec.

- Convert our RTT from milliseconds into seconds

Remember 1ms = 10^{-3}s:

```
window_seconds = 0.199 * 10^-3 = 0.000199
```

Our equivalent RTT window is 0.000199s

- Next we multiply the transfer rate by the length of our RTT window (in seconds)

(The unit analysis for this is `GB/s * s` leaving us with `GB`)

```
BDP = rx_rate_GB * window_seconds = 0.125 * 0.000199 = 0.000024875
```

Our BDP is 0.000024875GB.

- Convert 0.000024875GB to bytes:

Remember 1GB = 10^{9}B

```
BDP_bytes = 0.000024875 * 10^9 = 24875.0
```

Our BDP is 24875 bytes (or about 24.3KiB)

The `bitmath` way

All of this math can be done much quicker (and with greater accuracy) using the *`bitmath`* library. Let's see how:

```
>>> from bitmath import GB

>>> tx = 1/8.0

>>> rtt = 0.199 * 10**-3

>>> bdp = (GB(tx * rtt)).to_Byte()

>>> print bdp.to_KiB()

KiB(24.2919921875)
```

Note: To avoid integer rounding during division, don't forget to divide by `8.0` rather than `8`

We could shorten that even further:

```
>>> print (GB((1/8.0) * (0.199 * 10**-3))).to_Byte()
24875.0Byte
```

Get the current kernel parameters

Important to note is that the **per-socket** buffer sizes must not exceed the **core network** buffer sizes. Lets fetch our current core buffer sizes:

```
$ sysctl net.core.rmem_max net.core.wmem_max
net.core.rmem_max = 212992
net.core.wmem_max = 212992
```

Recall, these values are in bytes. What are they in KiB?

```
>>> print Byte(212992).to_KiB()
KiB(208.0)
```

This means our core networking buffer sizes are set to 208KiB each. Now let's check our current per-socket buffer sizes:

```
$ sysctl net.ipv4.tcp_rmem net.ipv4.tcp_wmem
net.ipv4.tcp_rmem = 4096        87380   6291456
net.ipv4.tcp_wmem = 4096        16384   4194304
```

Let's double-check that our buffer sizes aren't already out of wack (per-socket should be <= networking core)

```
>>> net_core_max = KiB(bytes=212992)

>>> ipv4_tcp_rmem_max = KiB(bytes=6291456)

>>> ipv4_tcp_rmem_max > net_core_max

True
```

It appears that my buffers aren't sized appropriately. We'll fix that when we set the tunable parameters.

Finally, how large is the entire system TCP buffer?

```
$ sysctl net.ipv4.tcp_mem
net.ipv4.tcp_mem = 280632       374176  561264
```

Our max system TCP buffer size is set to **561264**. Recall that this parameter is measured in **memory pages**. Most of the time your page size is `4096 bytes`, but you can check by running the command: `getconf PAGESIZE`. To convert the system TCP buffer size (561264) into a byte-based unit, we'll multiply it by our pagesize (4096):

```
>>> sys_pages = 561264

>>> page_size = 4096

>>> sys_buffer = Byte(sys_pages * page_size)

>>> print sys_buffer.to_MiB()

2192.4375MiB

>>> print sys_buffer.to_GiB()

2.14105224609GiB
```

The system max TCP buffer size is about 2.14GiB.

In review, we discovered the following:

- Our **core network** buffer size is insufficient (**212992**), we'll set it higher
- Our current **per-socket** buffer sizes are **6291456** and **4194304**

And we calculated the following:

- Our ideal **max** per-socket buffer size is **24875** bytes
- Our ideal **default** per-socket buffer size (half the **max**): **12437**

Finally: Set the new kernel parameters

Set the **core-network** buffer sizes:

```
$ sudo sysctl net.core.rmem_max=24875  net.core.wmem_max=24875
net.core.rmem_max = 4235
net.core.wmem_max = 4235
```

Set the **per-socket** buffer sizes:

```
$ sudo sysctl net.ipv4.tcp_rmem="4096 12437 24875" net.ipv4.tcp_wmem="4096 12437 24875"
net.ipv4.tcp_rmem = 4096 12437 24875
net.ipv4.tcp_wmem = 4096 12437 24875
```

And it's done! Testing this is left as an exercise for the reader. Note that in my experience this is less useful on wireless connections.

2.6.5 Creating Download Progress Bars

```
#!/usr/bin/env python
from __future__ import print_function
import logging
import time
import bitmath
import bitmath.integrations
import argparse
import requests
import progressbar
import os
import tempfile
import atexit
import random

# Files of various sizes to use in the demo.
#
# Moar here: https://www.kernel.org/pub/linux/kernel/v3.0/?C=S;O=D
REMOTES = [
    # patch-3.0.70.gz         20-Mar-2013 20:02  1.0M
    'https://www.kernel.org/pub/linux/kernel/v3.0/patch-3.4.92.xz',

    # patch-3.16.gz           03-Aug-2014 22:39  8.0M
    'https://www.kernel.org/pub/linux/kernel/v3.0/patch-3.16.gz',

    # patch-3.2.gz            05-Jan-2012 00:43   22M
    'https://www.kernel.org/pub/linux/kernel/v3.0/patch-3.2.gz',
]

######################################################################
p = argparse.ArgumentParser(description='bitmath demo suite')
p.add_argument('-d', '--down', help="Download Rate",
               type=bitmath.integrations.BitmathType,
               default=bitmath.MiB(4))

p.add_argument('-s', '--slowdown',
               help='Randomly pause to slow down the transfer rate',
               action='store_true', default=False)

args = p.parse_args()

######################################################################
# Save our example files somewhere. And then clean up every trace that
# anything every happened there. shhhhhhhhhhhhhhh
DESTDIR = tempfile.mkdtemp('demosuite', 'bitmath')
@atexit.register
def cleanup():
```

```
    for f in os.listdir(DESTDIR):
        os.remove(os.path.join(DESTDIR, f))
    os.rmdir(DESTDIR)

########################################################################
for f in REMOTES:
    print("""
########################################################################""")
    fname = os.path.basename(f)
    # An array of widgets to design our progress bar. Note how we use
    # BitmathFileTransferSpeed
    widgets = ['Bitmath Demo Suite (%s): ' % fname,
               progressbar.Percentage(), ' ',
               progressbar.Bar(marker=progressbar.RotatingMarker()), ' ',
               progressbar.ETA(), ' ',
               bitmath.integrations.BitmathFileTransferSpeed()]

    # The 'stream' keyword lets us http GET files in
    # chunks. http://docs.python-requests.org/en/latest/user/quickstart/#raw-response-content
    r = requests.get(f, stream=True)
    # We haven't began receiving the payload content yet, we have only
    # just received the response headers. Of interest is the
    # 'content-length' header which describes our payload in bytes
    #
    # http://bitmath.readthedocs.org/en/latest/classes.html#bitmath.Byte
    size = bitmath.Byte(int(r.headers['Content-Length']))

    # Demonstrate 'with' context handler, allowing us to customize all
    # bitmath string printing within the indented block. We don't need
    # all that precision anyway, just two points should do.
    #
    # http://bitmath.readthedocs.org/en/latest/module.html#bitmath-format
    with bitmath.format("{value:.2f} {unit}"):
        print("Downloading %s (%s) in %s chunks" % (f,
                                                    size.best_prefix(),
                                                    args.down.best_prefix()))

    # We have to save these files somewhere
    save_path = os.path.join(DESTDIR, fname)
    print("Saving to: %s" % save_path)
    print("")

    # OK. Let's create our actual progress bar now. See the 'maxval'
    # keyword? That's the size of our payload in bytes.
    pbar = progressbar.ProgressBar(
        widgets=widgets,
        maxval=int(size)).start()

    ####################################################################
    # Open a new file for binary writing and write 'args.down' size
    # chunks into it until we've received the entire payload
    with open(save_path, 'wb') as fd:
        # The 'iter_content' method accepts integer values of
        # bytes. Lucky for us, 'args.down' is a bitmath instance and
        # has a 'bytes' attribute we can feed into the method call.
        for chunk in r.iter_content(int(args.down.bytes)):
            fd.write(chunk)
            # The progressbar will end the entire cosmos as we know it
```

```
            # if we try to .update() it beyond it's MAXVAL
            # parameter.
            #
            # That's something I'd like to avoid taking the
            # responsibility for.
            if (pbar.currval + args.down.bytes) < pbar.maxval:
                pbar.update(pbar.currval + int(args.down.bytes))

            # We can add an pause to artificially speed up/slowdown
            # the transfer rate. Allows us to see different units.
            if args.slowdown:
                # randomly slow down 1/5 of the time
                if random.randrange(0, 100) % 5 == 0:
                    time.sleep(random.randrange(0, 500) * 0.01)

    # Nothing to see here. Go home.
    pbar.finish()

########################################################################
print("""
########################################################################
List downloaded contents
* Filter for .xz files only
""")

for p,bm in bitmath.listdir(DESTDIR,
                            filter='*.xz'):
    print(p, bm)

########################################################################
print("""
########################################################################
List downloaded contents
* Filter for .gz files only
* Print using best human readable prefix
""")

for p,bm in bitmath.listdir(DESTDIR,
                            filter='*.gz',
                            bestprefix=True):
    print(p, bm)

########################################################################
print("""
########################################################################
List downloaded contents
* No filter set, to display all files
* Limit precision of printed file size to 3 digits
* Print using best human readable prefix
""")

for p,bm in bitmath.listdir(DESTDIR,
                            bestprefix=True):
    with bitmath.format("{value:.3f} {unit}"):
        print(p, bm)

########################################################################
print("""
```

```
########################################################################
Sum the size of all downloaded files together
* Print with best prefix and 3 digits of precision
""")

discovered_files = [f[1] for f in bitmath.listdir(DESTDIR)]
total_size = reduce(lambda x,y: x+y, discovered_files).best_prefix().format("{value:.3f} {unit}")
print("Total size of %s downloaded items: %s" % (len(discovered_files), total_size))
```

- View the the source for the demo suite on GitHub

2.6.6 Reading a Devices Storage Capacity

Important: Superuser (root/admin) privileges are required to allow *bitmath.query_device_capacity()* to make the low-level system calls to read a devices capacity. Use of this function on a device the user does not have access to will result in run-time errors.

Using *bitmath.query_device_capacity()* we can read the size of a storage device or a partition on a device.

Examples of supported devices include:

- Standard Hard Drives/External Drives
- Filesystem Partitions
- Loop Devices
- LVM Logical Volumes
- Encrypted LUKS Volumes
- iSCSI Devices

Usage is fairly straight-forward. Create an open file handle of the device you want to read the capacity of and then create a bitmath object with the `query_device_capacity` function. Here's an example where we read the capacity of device `sda`, the first device on the example system.

```
>>> import bitmath
>>> fh = open('/dev/sda', 'r')
>>> sda_capacity = bitmath.query_device_capacity(fh)
>>> fh.close()
>>> print sda_capacity.best_prefix()
238.474937439 GiB
```

We can simplify this so that the file handle is automatically closed for us by using the `with` context manager.

```
>>> with open('/dev/sda', 'r') as fh:
...     sda_capacity = bitmath.query_device_capacity(fh)
>>> print sda_capacity.best_prefix()
238.474937439 GiB
```

2.7 Contributing to bitmath

This section describes the guidelines for contributing to bitmath.

2.7.1 Issue Reporting

If you are encounter an issue with the bitmath library, please use the following template when describing your issue:

```
Description of the issue (include full error messages):

How to reproduce the issue:

How reproducible (every time? intermittently?):

Version of bitmath effected (git hashes are OK):

Your operating system and Python release-version:

What you expected to happen:

What actually happened:
```

- Open a new issue
- View open issues

2.7.2 Code Style/Formatting

Please conform to **PEP 0008** for code formatting. This specification outlines the style that is required for patches.

Your code must follow this (or note why it can't) before patches will be accepted. There is one consistent exception to this rule:

E501 Line too long

The `pep8` tests for bitmath include a `--ignore` option to automatically exclude **E501** errors from the tests.

2.7.3 Commit Messages

Please write intelligent commit messages.

For example:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as
```

```
the subject of an email and the rest of the text as the body.  The
blank line separating the summary from the body is critical (unless
you omit the body entirely); tools like rebase can get confused if
you run the two together.

Write your commit message in the imperative: "Fix bug" and not
"Fixed bug" or "Fixes bug."  This convention matches up with commit
messages generated by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed
  by a single space, with blank lines in between, but conventions
  vary here

- Use a hanging indent
```

2.7.4 Pull Requests

After a pull request is submitted on GitHub two automatic processes are started:

1. Travis-CI clones the new pull request and runs the *automated test suite*.
2. Coveralls clones the new pull request and determines if the request would increase or decrease the overall code test coverage.

Please check back shortly after submitting a pull request to verify that the Travis-CI process passes.

What Happens If The Build Breaks

Pull requests which break the build will be looked at closely and you may be asked to fix the tests.

The bitmath project **welcomes all contributors** so **it's OK** if you're unable to fix the tests yourself. Just leave a comment in the pull request explaining so if that is the case.

Likewise, if Coveralls indicates the pull request would decrease the overall test-coverage, and you aren't able to fix it yourself, just leave a comment in the pull request.

2.7.5 Automated Tests

Write unittests for any new functionality, *if you are up to the task*. This is not a requirement, but it does get you a lot of karma.

All bitmath code includes unit tests to verify expected functionality. In the rest of this section we'll learn how the unit tests are put together and how to interact with them.

Components

bitmath unit tests are integrated with/depend on the following items:

- Travis CI - Free online service providing *continuous integration* functionality for open source projects. Tests are ran automatically on every git commit. Integrates with GitHub to notify you if a pull request passes or fails all unitests.

- Coveralls - Free online service providing code test coverage reporting. Integrates with GitHub to notify you if a pull-request would improve/decrease overall code test coverage.
- unittest - Python unit testing framework. All bitmath tests are written using this framework.
- nose - Per the **nose** website: "*extends unittest to make testing easier*". **nose** is used to run our unit tests.
- coverage - A tool for measuring code coverage of Python programs. For bitmath we require a minimum test coverage of **90%**. This is invoked by **nose** automatically.
- pep8 - A tool to check Python code against some of the style conventions in **PEP 0008**.
- pyflakes - A simple program which checks Python source files for errors.
- virtualenv - A tool to create isolated Python environments. Allows us to install additional package dependencies without requiring access to the system site-packages directory.
- Makefiles - Utility scripts used for project building and testing. How bitmath uses **Makefiles** is described later in this section.

Targets

In the scope of this document, we use the term *target* in the context of *makefile targets*. For the purpose of this documentation, we can think of these *targets* as pre-defined commands coded in a makefile. bitmath testing targets include:

- `ci` - Run the tests exactly how they are ran in Travis-CI. The `ci` target automatically calls the `pep8`, `pyflakes`, `uniquetestnames`, and `unittests` targets.
- `ci3` - Is the same as the `ci` target, except it runs using the Python 3.x interpreter.
- `unittests` - Run the functional test suite.
- `pep8` - Run **PEP 0008** syntax checks.
- `pyflakes` - Run *pyflakes* error checks.
- `clean` - Remove temporary files and build artifacts from the checked-out repository.
- `uniquetestnames` - Ensures no unit tests have the same name.
- `tests` - A quicker version of `ci`. Different from `ci` in that `tests` uses libraries installed on the local development workstation. `tests` runs the `unittests`, `pep8`, `uniquetestnames`, and `pyflakes` tests automatically.

To ensure the highest degree of confidence in test results you should **always use** the `ci` and `ci3` targets.

When Travis-CI runs an integration test, it calls the `ci` and `ci3` targets.

Running the Tests

The bitmath test suite is invoked via the Makefile. The following is an example of how to run the `ci` test target manually:

```
[~/Projects/bitmath] 17:22:21  (master)
$ make ci
##########################################
# Running Unique TestCase checker
##########################################
./tests/test_unique_testcase_names.sh
##########################################
# Creating a virtualenv
```

```
9  ##########################################
10 virtualenv bitmathenv
11 New python executable in bitmathenv/bin/python
12 Installing setuptools, pip...done.
13 . bitmathenv/bin/activate && pip install -r requirements.txt
14 Downloading/unpacking python-coveralls (from -r requirements.txt (line 1))
15   Downloading python_coveralls-2.4.3-py2.py3-none-any.whl
16 Downloading/unpacking nose (from -r requirements.txt (line 2))
17
18 ... snip ...
19
20 Convert a bitmath GiB into a Tb ... ok
21 Convert a bitmath PiB into a TiB ... ok
22 Convert a bitmath GiB into a Tib ... ok
23 Convert to kb ... ok
24 Convert a bitmath Bit into a MiB ... ok
25 bitmath type converted to the same unit is properly converted ... ok
26 float(bitmath) returns a float ... ok
27 int(bitmath) returns an int ... ok
28 long(bitmath) returns a long ... ok
29
30 Name      Stmts   Miss  Cover   Missing
31 -------------------------------------
32 bitmath     440      1    99%   1152
33 ----------------------------------------------------------------------
34 Ran 163 tests in 0.035s
35
36 OK
37 :
```

On line **2** we see how to call a makefile target. In this case it's quite straightforward: `make ci`. Other targets are called in the same way. For example, to run the `clean` target, you run the command `make clean`. To run the Python 3.x test suite, you would run the command `make ci3`.

Troubleshooting

If you find yourself unable to run the unit tests:

1. Search for relevant error messages
2. **Read** the error message closely. The solution could be hidden in the error message output. The problem could be as simple as a missing dependency
3. If you are unable to figure out all the necessary dependencies to run the tests, file an issue on that specific projects GitHub issue tracker. Include the full error message.

2.8 Appendices

2.8.1 Rules for Math

This section describes what we need to know to effectively use bitmath for arithmetic. Because bitmath allows the use of instances as operands on either side of the operator it is especially important to understand their behavior. Just as in normal every-day math, not all operations yield the same result if the operands are switched. E.g., `1 - 2 = -1` whereas `2 - 1 = 1`.

This section includes discussions of the results for each supported **mixed math** operation. For mixed math operations (i.e., an operation with a bitmath instance and a number type), implicit coercion **may** happen. That is to say, a bitmath instance will be converted to a number type.

When coercion happens is determined by the following conditions and rules:

1. Precedence and Associativity of Operators in Python[1]
2. Situational semantics – some operations, though mathematically valid, do not make logical sense when applied to context.

Terminology

The definitions describes some of the terminology used throughout this section.

Coercion The act of converting operands into a common type to support arithmetic operations. Somewhat similar to how adding two fractions requires coercing each operand into having a common denominator.

Specific to the bitmath domain, this concept means using an instance's prefix value for mixed-math.

Operand The object(s) of a mathematical operation. That is to say, given `1 + 2`, the operands would be **1** and **2**.

Operator The mathematical operation to evaluate. Given `1 + 2`, the operation would be addition, **+**.

LHS *Left-hand side*. In discussion this specifically refers to the operand on the left-hand side of the operator.

RHS *Right-hand side*. In discussion this specifically refers to the operand on the right-hand side of the operator.

Two bitmath operands

This section describes what happens when two bitmath instances are used as operands. There are three possible results from this type of operation.

Addition and subtraction The result will be of the type of the LHS.

[1] For a less technical review of precedence and associativity, see Programiz: Precedence and Associativity of Operators in Python

Multiplication Supported, but yields strange results.

```
In [10]: first = MiB(5)

In [11]: second = kB(2)

In [12]: first * second
Out[12]: MiB(10000.0)

In [13]: (first * second).best_prefix()
Out[13]: GiB(9.765625)
```

As we can see on lines **6** and **9**, multiplying even two relatively small quantities together (`MiB(5)` and `kB(2)`) yields quite large results.

Internally, this is implemented as:

$$(5 \cdot 2^{20}) \cdot (2 \cdot 10^3) = 10,485,760,000B$$

$$10,485,760,000B \cdot \frac{1MiB}{1,048,576B} = 10,000MiB$$

Division The result will be a number type due to unit cancellation.

Mixed Types: Addition and Subtraction

This describes the behavior of addition and subtraction operations where one operand is a bitmath type and the other is a number type.

Mixed-math addition and subtraction **always** return a type from the `numbers` family (integer, float, long, etc...). This rule is true regardless of the placement of the operands, with respect to the operator.

Discussion: Why do `100 - KiB(90)` and `KiB(100) - 90` both yield a result of `10.0` and not another bitmath instance, such as `KiB(10.0)`?

When implementing the math part of the object datamodel customizations[2] there were two choices available:

1. Offer no support at all. Instead raise a `NotImplemented` exception.
2. Consistently apply coercion to the bitmath operand to produce a useful result (*useful* if you know the rules of the library).

In the end it became a philosophical decision guided by scientific precedence.

Put simply, bitmath uses the significance of the **least significant operand**, specifically the number-type operand because it lacks semantic significance. In application this means that we drop the semantic significance of the bitmath operand. That is to say, given an input like `GiB(13.37)` (equivalent to == 13.37 * 2^{30}), the only property used in calculations is the prefix value, `13.37`.

Numbers carry mathematical significance, in the form of precision, but what they lack is *semantic* (contextual) significance. A number by itself is just a measurement of an arbitrary quantity of *stuff*. In mixed-type math, bitmath effectively treats numbers as mathematical constants.

A bitmath instance also has mathematical significance in that an instance is a measurement of a quantity (bits in this case) and that quantity has a measurable precision. But a bitmath instance is more than just a measurement, it is a specialized representation of a count of bits. This gives bitmath instances *semantic* significance as well.

And so, in deciding how to handle mixed-type (really what we should say is mixed-significance) operations, we chose to model the behavior off of an already established set of rules. Those rules are the Rules of Significance Arithmetic[3].

[2] Python Datamodel Customization Methods
[3] https://en.wikipedia.org/wiki/Significance_arithmetic

Let's look at an example of this in action:

```
In [8]: num = 42

In [9]: bm = PiB(24)

In [10]: print num + bm
66.0
```

Equivalently, divorcing the bitmath instance from it's value (this is coercion):

```
In [12]: bm_value = bm.value

In [13]: print num + bm_value
66.0
```

What it all boils down to is this: if we don't provide a unit then bitmath won't give us one back. There is no way for bitmath to guess what unit the operand was *intended* to carry. Therefore, the behavior of bitmath is **conservative**. It will meet us half way and do the math, but it will not return a unit in the result.

Mixed Types: Multiplication and Division

Multiplication has *commutative* properties. This means that the ordering of the operands is **not significant**. Because of this fact bitmath allows arbitrary placement of the operands, treating the numeric operand as a constant. Here's an example demonstrating this.

```
In [2]: 10 * KiB(43)
Out[2]: KiB(430.0)

In [3]: KiB(43) * 10
Out[3]: KiB(430.0)
```

Division, however, *does not* have this commutative property. I.e., the placement of the operands **is** significant. Additionally, there is a semantic difference in division. Dividing a quantity (e.g. `MiB(100)`) by a constant (`10`) makes complete sense. Conceptually (in the domain of bitmath), the intention of `MiB(100) / 10)` is to separate `MiB(10)` into **10** equal sized parts.

```
In [4]: KiB(43) / 10
Out[4]: KiB(4.2998046875)
```

The reverse operation does not maintain semantic validity. Stated differently, it does not make logical sense to divide a constant by a measured quantity of *stuff*. If you're still not clear on this, ask yourself what you would expect to get if you did this:

$$\frac{100}{kB(33)} = x$$

Footnotes

2.8.2 On Units

As previously stated, in this module you will find two very similar sets of classes available. These are the **NIST** and **SI** prefixes. The **NIST** prefixes are all base 2 and have an 'i' character in the middle. The **SI** prefixes are base 10 and have no 'i' character.

For smaller values, these two systems of unit prefixes are roughly equivalent. The `round()` operations below demonstrate how close in a percent one "unit" of SI is to one "unit" of NIST.

```
In [15]: one_kilo = 1 * 10**3

In [16]: one_kibi = 1 * 2**10

In [17]: round(one_kilo / float(one_kibi), 2)

Out[17]: 0.98

In [18]: one_tera = 1 * 10**12

In [19]: one_tebi = 1 * 2**40

In [20]: round(one_tera / float(one_tebi), 2)

Out[20]: 0.91

In [21]: one_exa = 1 * 10**18

In [22]: one_exbi = 1 * 2**60

In [23]: round(one_exa / float(one_exbi), 2)

Out[23]: 0.87
```

They begin as roughly equivalent, however as you can see (lines: **7**, **15**, and **23**), they diverge significantly for higher values.

Why two unit systems? Why take the time to point this difference out? Why should you care? The Linux Documentation Project comments on that:

> Before these binary prefixes were introduced, it was fairly common to use k=1000 and K=1024, just like b=bit, B=byte. Unfortunately, the M is capital already, and cannot be capitalized to indicate binary-ness.
>
> At first that didn't matter too much, since memory modules and disks came in sizes that were powers of two, so everyone knew that in such contexts "kilobyte" and "megabyte" meant 1024 and 1048576 bytes, respectively. What originally was a sloppy use of the prefixes "kilo" and "mega" started to become regarded as the "real true meaning" when computers were involved. But then disk technology changed, and disk sizes became arbitrary numbers. After a period of uncertainty all disk manufacturers settled on the standard, namely k=1000, M=1000k, G=1000M.
>
> The situation was messy: in the 14k4 modems, k=1000; in the 1.44MB diskettes, M=1024000; etc. In 1998 the IEC approved the standard that defines the binary prefixes given above, enabling people to be precise and unambiguous.
>
> Thus, today, MB = 1000000B and MiB = 1048576B.
>
> In the free software world programs are slowly being changed to conform. When the Linux kernel boots and says:

```
hda: 120064896 sectors (61473 MB) w/2048KiB Cache
```

> the MB are megabytes and the KiB are kibibytes.

- Source: `man 7 units` - http://man7.org/linux/man-pages/man7/units.7.html

Furthermore, to quote the National Institute of Standards and Technology (NIST):

> "Once upon a time, computer professionals noticed that 2^{10} was very nearly equal to 1000 and started using the SI prefix "kilo" to mean 1024. That worked well enough for a decade or two because everybody who talked kilobytes knew that the term implied 1024 bytes. But, almost overnight a much more numerous "everybody" bought computers, and the trade computer professionals needed to talk to physicists and

> engineers and even to ordinary people, most of whom know that a kilometer is 1000 meters and a kilogram is 1000 grams.
>
> "Then data storage for gigabytes, and even terabytes, became practical, and the storage devices were not constructed on binary trees, which meant that, for many practical purposes, binary arithmetic was less convenient than decimal arithmetic. The result is that today "everybody" does not "know" what a megabyte is. When discussing computer memory, most manufacturers use megabyte to mean 2^{20} = 1 048 576 bytes, but the manufacturers of computer storage devices usually use the term to mean 1 000 000 bytes. Some designers of local area networks have used megabit per second to mean 1 048 576 bit/s, but all telecommunications engineers use it to mean 106 bit/s. And if two definitions of the megabyte are not enough, a third megabyte of 1 024 000 bytes is the megabyte used to format the familiar 90 mm (3 1/2 inch), "1.44 MB" diskette. The confusion is real, as is the potential for incompatibility in standards and in implemented systems.
>
> "Faced with this reality, the IEEE Standards Board decided that IEEE standards will use the conventional, internationally adopted, definitions of the SI prefixes. Mega will mean 1 000 000, except that the base-two definition may be used (if such usage is explicitly pointed out on a case-by-case basis) until such time that prefixes for binary multiples are adopted by an appropriate standards body."

2.8.3 Who uses Bitmath

Shout-outs to all of the bitmath adopters out there I was able to identify:

- ClusterHQ's "Flocker". A data volume manager for Dockerized applications
- VMware's vsphere flocker storage driver
- EMC's scaleio flocker storage driver
- Dell Storage's storage center block device flocker driver
- TravelCRM - Free CRM for travel companies - Bitbucket
- direscraw by Brian Mikolajczyk for recovering lost files
- sizer by Calle Liljeholm. Calculating useable capacity in a flocker cluster

2.8.4 Related Projects

Bitmath is not the first project to tackle a challenge of this nature, handling units in a sane OOP approach. Several other Python libraries exist which provide similar functionality to bitmath. It only seems fair that we should point out these other libraries in case bitmath isn't the best fit for you.

Magnitude

> *Magnitude implements efficient computation with physical quantities. It allows you to do mathematical operations with them as if they were numbers, taking care of the units behind the scenes.*

Magnitide, from Juan Reyero, is a *very* extensible library for working with a large variety of units (e.g., `mile` = *one mile*), as well as derived units (e.g., `mile/hour`). Scaling, such as indicating one **mega** byte (`1 MB`) is also programmable with Magnitude. Juan is also kind enough to include a similar "related projects" section in his documentation.

- Magnitude GitHub Project
- Magnitude Docs

hurry.filesize

> *hurry.filesize a simple Python library that can take a number of bytes and returns a human-readable string with the size in it, in kilobytes (K), megabytes (M), etc.*

hurry.filesize is very limited in functionality when compared to the other alternatives. However, it is an extremely simple and lightweight module. If you're looking for a library just for turning counts of bytes into human-readable strings, then hurry.filesize will be great for you.

If you need any more functionality, such as greater control over *output formatting*, or *arithmetic calculations*, then you will find hurry.filesize lacking. This project has not updated since 2009, so I would not expect to see updates any time soon.

- PyPi Homepage & Download

SymPy - Units

> *This module provides around 200 predefined units that are commonly used in the sciences. Additionally, it provides the ''Unit'' class which allows you to define your own units.*

The **Units** module from the SymPy library is another option. Like **Magnitude**, the Units library is very extensible and includes around 200 built-in units by default. While technically it supports handling quantities such as `1337 PiB`, this support must be configured by the user.

In contrast, the bitmath module includes classes representing the full spectrum of byte and bit based units, out of the box. No conversion or derivation code required of the user.

- Units Homepage & Docs
- Download available through `pip`, or your distribution's package system

Unum

> *Unum stands for 'unit-numbers'. It is a Python module that allows to define and manipulate true quantities, i.e. numbers with units such as 60 seconds, [...], 30 dollars etc. The module validates unit consistency in arithmetic expressions; it provides also automatic conversion and output formatting. Unum is designed to be reliable, easy-to-use, customizable and open to any unit definition.*

Unum, by Pierre X. Denis, is another extensible library for unit manipulation. The module does not appear to have seen any activity in quite some time. Looking over the docs gives me the impression that it also has a tendency to pollute your namespace with objects like `M` and anything else it pre-defines.

- Unum Homepage and Docs
- Unum Source Download

2.9 NEWS

2.9.1 bitmath-1.3.1-1

bitmath-1.3.1-1 was published on 2016-07-17.

Changes

Added Functionality

- New function: `bitmath.parse_string_unsafe()`, a less strict version of `bitmath.parse_string()`. Accepts inputs using *non-standard* prefix units (such as single-letter, or mis-capitalized units).
- Inspired by @darkblaze69's request in #60 "Problems in parse_string".

Project

Ubuntu

- Bitmath is now available for installation via Ubuntu Xenial, Wily, Vivid, Trusty, and Precise PPAs.
- Ubuntu builds inspired by @hkraal reporting an installation issue on Ubuntu systems.

Documentation

- Cleaned up a lot of broken or re-directing links using output from the Sphinx `make linkcheck` command.

2.9.2 bitmath-1.3.0-1

bitmath-1.3.0-1 was published on 2016-01-08.

Changes

Bug Fixes

- Closed GitHub Issue #55 "best_prefix for negative values". Now `bitmath.best_prefix()` returns correct prefix units for negative values. Thanks mbdm!

2.9.3 bitmath-1.2.4-1

bitmath-1.2.4-1 was published on 2015-11-30.

Changes

Added Functionality

- New bitmath module function: *bitmath.query_device_capacity()*. Create *bitmath.Byte* instances representing the capacity of a block device. Support is presently limited to Linux and Mac.
- The *bitmath.parse_string()* function now can parse 'octet' based units. Enhancement requested in #53 parse french unit names by walidsa3d.

Bug Fixes

- #49 - Fix handling unicode input in the bitmath.parse_string function. Thanks drewbrew!
- #50 - Update the `setup.py` script to be python3.x compat. Thanks ssut!

Documentation

- The project documentation is now installed along with the bitmath library in RPM packages.

Project

Fedora/EPEL

Look for separate python3.x and python2.x packages coming soon to Fedora and EPEL. This is happening because of the initiative to update the base Python implementation on Fedora to Python 3.x

- BZ1282560

2.9.4 bitmath-1.2.3-1

bitmath-1.2.3-1 was published on 2015-01-03.

Changes

Added Functionality

- New utility: `progressbar` integration: bitmath.integrations.BitmathFileTransferSpeed. A more functional file transfer speed widget.

Documentation

- The command-line `bitmath` tool now has online documentation
- A full demo of the `argparse` and `progressbar` integrations has been written. Additionally, it includes a comprehensive demonstration of the full capabilities of the bitmath library. View it in the *Real Life Demos* Creating Download Progress Bars example.

Project

Tests

- Travis-CI had some issues with installing dependencies for the 3.x build unittests. These were fixed and the build status has returned back to normal.

2.9.5 bitmath-1.2.0-1

bitmath-1.2.0-1 was published on 2014-12-29.

Changes

Added Functionality

- New utility: `argparse` integration: bitmath.BitmathType. Allows you to specify arguments as bitmath types.

Documentation

- The command-line `bitmath` tool now has a proper manpage

Project

Tests

- The command-line `bitmath` tool is now properly unittested. Code coverage back to ~100%.

2.9.6 bitmath-1.1.0-1

bitmath-1.1.0-1 was published on 2014-12-20.

- GitHub Milestone Tracker for 1.1.0

Changes

Added Functionality

- New `bitmath` command-line tool added. Provides CLI access to basic unit conversion functions
- New utility function bitmath.parse_string for parsing a human-readable string into a bitmath object. Patch submitted by new contributor tonycpsu.

2.9.7 bitmath-1.0.5-1 through 1.0.8-1

bitmath-1.0.8-1 was published on 2014-08-14.

- GitHub Milestone Tracker for 1.0.8

Major Updates

- bitmath has a proper documentation website up now on Read the Docs, check it out: bitmath.readthedocs.io
- bitmath is now Python 3.x compatible
- bitmath is now included in the Extra Packages for Enterprise Linux EPEL6 and EPEL7 repositories (pkg info)
- merged 6 pull requests from 3 contributors

Bug Fixes

- fixed some math implementation bugs
  - commutative multiplication
  - true division

Changes

Added Functionality

- best-prefix guessing: automatic best human-readable unit selection
- support for bitwise operations
- formatting customization methods (including plural/singular selection)
- exposed many more instance attributes (all instance attributes are usable in custom formatting)
- a context manager for applying formatting to an entire block of code
- utility functions for sizing files and directories
- add instance properties equivalent to `instance.to_THING()` methods

Project

Tests

- Test suite is now implemented using Python virtualenv's for consistency across across platforms
- Test suite now contains 150 unit tests. This is **110** more tests than the previous major release (1.0.4-1)
- Test suite now runs on EPEL6 and EPEL7
- Code coverage is stable around 95-100%

2.9.8 bitmath-1.0.4-1

This is the first release of **bitmath**. bitmath-1.0.4-1 was published on 2014-03-20.

Project

Available via:

- PyPi
- Fedora 19
- Fedora 20

bitmath had been under development for 12 days when the 1.0.4-1 release was made available.

Debut Functionality

- Converting between **SI** and **NIST** prefix units (`GiB` to `kB`)
- Converting between units of the same type (SI to SI, or NIST to NIST)
- Basic arithmetic operations (subtracting 42KiB from 50GiB)
- Rich comparison operations (`1024 Bytes == 1KiB`)
- Sorting
- Useful *console* and *print* representations

2.10 Contact

Hi, I'm Tim Bielawa, the bitmath maintainer. Would you like to get in touch? Maybe you want to peek at other stuff I'm working on? Go right ahead:

Blog I maintain a blog which mostly covers technical problems I encounter that I think are interesting and how I solved them. Lots of rich commentary and details included!

- Technitribe

Open Source Almost every project I work on, code or not, ends up on GitHub eventually. You can see what else I've been busy with on my profile.

- GitHub: tbielawa

Tweet-Tweet I have been known to tweet from time to time.

- @tbielawa

Bugs/Issues/Requests All contributions related directly to the bitmath project, i.e. bug reports or feature requests, should be posted to the project issue tracker on GitHub. Please see the Contributing section for more information.

- *Contributing*

Saying hello I'm on the freenode IRC network Monday through Friday, from around 9am EST through 5pm EST.

- Issue a `/who tbielawa*` command to the server, a handle with the netmask `~tbielawa@redhat/tbielawa` will appear for you to `/query` if I'm online.

E-Mail If you want to contact me directly, clone the project and look at any of my bitmath commits to find my email address.

2.11 Copyright

2.12 Examples

2.12.1 Arithmetic

```
>>> import bitmath
>>> log_size = bitmath.kB(137.4)
>>> log_zipped_size = bitmath.Byte(987)
>>> print "Compression saved %s space" % (log_size - log_zipped_size)
Compression saved 136.413kB space
>>> thumb_drive = bitmath.GiB(12)
>>> song_size = bitmath.MiB(5)
>>> songs_per_drive = thumb_drive / song_size
>>> print songs_per_drive
2457.6
```

2.12.2 Convert Units

File size unit conversion:

```
>>> from bitmath import *
>>> dvd_size = GiB(4.7)
>>> print "DVD Size in MiB: %s" % dvd_size.to_MiB()
DVD Size in MiB: 4812.8 MiB
```

2.12.3 Select a human-readable unit

```
>>> small_number = kB(100)
>>> ugly_number = small_number.to_TiB()

>>> print ugly_number
9.09494701773e-08 TiB
>>> print ugly_number.best_prefix()
97.65625 KiB
```

2.12.4 Rich Comparison

```
>>> cd_size = MiB(700)
>>> cd_size > dvd_size
False
>>> cd_size < dvd_size
True
>>> MiB(1) == KiB(1024)
True
```

```
>>> MiB(1) <= KiB(1024)
True
```

2.12.5 Sorting

```
>>> sizes = [KiB(7337.0), KiB(1441.0), KiB(2126.0), KiB(2178.0),
                 KiB(2326.0), KiB(4003.0), KiB(48.0), KiB(1770.0),
                 KiB(7892.0), KiB(4190.0)]

>>> print sorted(sizes)
[KiB(48.0), KiB(1441.0), KiB(1770.0), KiB(2126.0), KiB(2178.0),
KiB(2326.0), KiB(4003.0), KiB(4190.0), KiB(7337.0), KiB(7892.0)]
```

2.12.6 Custom Formatting

- Use of the custom formatting system
- All of the available instance properties

Example:

```
>>> longer_format = """Formatting attributes for %s
   ...: This instances prefix unit is {unit}, which is a {system} type unit
   ...: The unit value is {value}
   ...: This value can be truncated to just 1 digit of precision: {value:.1f}
   ...: In binary this looks like: {binary}
   ...: The prefix unit is derived from a base of {base}
   ...: Which is raised to the power {power}
   ...: There are {bytes} bytes in this instance
   ...: The instance is {bits} bits large
   ...: bytes/bits without trailing decimals: {bytes:.0f}/{bits:.0f}""" % str(ugly_number)

>>> print ugly_number.format(longer_format)
Formatting attributes for 5.96046447754 MiB
This instances prefix unit is MiB, which is a NIST type unit
The unit value is 5.96046447754
This value can be truncated to just 1 digit of precision: 6.0
In binary this looks like: 0b10111110101111000010000000
The prefix unit is derived from a base of 2
Which is raised to the power 20
There are 6250000.0 bytes in this instance
The instance is 50000000.0 bits large
bytes/bits without trailing decimals: 6250000/50000000
```

2.12.7 Utility Functions

bitmath.getsize()

```
>>> print bitmath.getsize('python-bitmath.spec')
3.7060546875 KiB
```

bitmath.parse_string()

Parse a string with standard units:

```
>>> import bitmath
>>> a_dvd = bitmath.parse_string("4.7 GiB")
>>> print type(a_dvd)
<class 'bitmath.GiB'>
>>> print a_dvd
4.7 GiB
```

bitmath.parse_string_unsafe()

Parse a string with ambiguous units:

```
>>> import bitmath
>>> a_gig = bitmath.parse_string_unsafe("1gb")
>>> print type(a_gig)
<class 'bitmath.GB'>
>>> a_gig == bitmath.GB(1)
True
>>> bitmath.parse_string_unsafe('1gb') == bitmath.parse_string_unsafe('1g')
True
```

bitmath.query_device_capacity()

```
>>> import bitmath
>>> with open('/dev/sda') as fp:
...     root_disk = bitmath.query_device_capacity(fp)
...     print root_disk.best_prefix()
...
238.474937439 GiB
```

bitmath.listdir()

```
>>> for i in bitmath.listdir('./tests/', followlinks=True, relpath=True, bestprefix=True):
...     print i
...
('tests/test_file_size.py', KiB(9.2900390625))
('tests/test_basic_math.py', KiB(7.1767578125))
('tests/__init__.py', KiB(1.974609375))
('tests/test_bitwise_operations.py', KiB(2.6376953125))
('tests/test_context_manager.py', KiB(3.7744140625))
('tests/test_representation.py', KiB(5.2568359375))
('tests/test_properties.py', KiB(2.03125))
('tests/test_instantiating.py', KiB(3.4580078125))
('tests/test_future_math.py', KiB(2.2001953125))
('tests/test_best_prefix_BASE.py', KiB(2.1044921875))
('tests/test_rich_comparison.py', KiB(3.9423828125))
('tests/test_best_prefix_NIST.py', KiB(5.431640625))
('tests/test_unique_testcase_names.sh', Byte(311.0))
('tests/.coverage', KiB(3.1708984375))
('tests/test_best_prefix_SI.py', KiB(5.34375))
('tests/test_to_built_in_conversion.py', KiB(1.798828125))
('tests/test_to_Type_conversion.py', KiB(8.0185546875))
('tests/test_sorting.py', KiB(4.2197265625))
('tests/listdir_symlinks/10_byte_file_link', Byte(10.0))
('tests/listdir_symlinks/depth1/depth2/10_byte_file', Byte(10.0))
('tests/listdir_nosymlinks/depth1/depth2/10_byte_file', Byte(10.0))
('tests/listdir_nosymlinks/depth1/depth2/1024_byte_file', KiB(1.0))
('tests/file_sizes/kbytes.test', KiB(1.0))
('tests/file_sizes/bytes.test', Byte(38.0))
('tests/listdir/10_byte_file', Byte(10.0))
```

2.12.8 Formatting

```
>>> with bitmath.format(fmt_str="[{value:.3f}@{unit}]"):
...     for i in bitmath.listdir('./tests/', followlinks=True, relpath=True, bestprefix=True):
...         print i[1]
...
[9.290@KiB]
[7.177@KiB]
[1.975@KiB]
[2.638@KiB]
[3.774@KiB]
[5.257@KiB]
[2.031@KiB]
[3.458@KiB]
[2.200@KiB]
[2.104@KiB]
[3.942@KiB]
[5.432@KiB]
[311.000@Byte]
[3.171@KiB]
[5.344@KiB]
[1.799@KiB]
[8.019@KiB]
[4.220@KiB]
[10.000@Byte]
[10.000@Byte]
[10.000@Byte]
[1.000@KiB]
[1.000@KiB]
[38.000@Byte]
[10.000@Byte]
```

2.12.9 `argparse` Integration

Example script using `bitmath.integrations.BitmathType` as an argparser argument type:

```
import argparse
import bitmath
parser = argparse.ArgumentParser(
    description="Arg parser with a bitmath type argument")
parser.add_argument('--block-size',
                    type=bitmath.integrations.BitmathType,
                    required=True)

results = parser.parse_args()
print "Parsed in: {PARSED}; Which looks like {TOKIB} as a Kibibit".format(
    PARSED=results.block_size,
    TOKIB=results.block_size.Kib)
```

If ran as a script the results would be similar to this:

```
$ python ./bmargparse.py --block-size 100MiB
Parsed in: 100.0 MiB; Which looks like 819200.0 Kib as a Kibibit
```

2.12.10 `progressbar` Integration

Use `bitmath.integrations.BitmathFileTransferSpeed` as a `progressbar` file transfer speed widget to monitor download speeds:

```
import requests
import progressbar
import bitmath
import bitmath.integrations

FETCH = 'https://www.kernel.org/pub/linux/kernel/v3.0/patch-3.16.gz'
widgets = ['Bitmath Progress Bar Demo: ', ' ',
           progressbar.Bar(marker=progressbar.RotatingMarker()), ' ',
           bitmath.integrations.BitmathFileTransferSpeed()]

r = requests.get(FETCH, stream=True)
size = bitmath.Byte(int(r.headers['Content-Length']))
pbar = progressbar.ProgressBar(widgets=widgets, maxval=int(size),
                               term_width=80).start()
chunk_size = 2048
with open('/dev/null', 'wb') as fd:
    for chunk in r.iter_content(chunk_size):
        fd.write(chunk)
        if (pbar.currval + chunk_size) < pbar.maxval:
            pbar.update(pbar.currval + chunk_size)
pbar.finish()
```

If ran as a script the results would be similar to this:

```
$ python ./smalldl.py
Bitmath Progress Bar Demo:  |||||||||||||||||||||||||||||||||||||||||||||||||| 1.58 MiB/s
```

b

Symbols

A

B

E

F

G

K

L

M

N

P

Q

S

T

U

V

Y

Z