Adding Abseil (absl) flags quickstart

WARNING This module is deprecated. We no long use it in new models and your projects should not depend on it. We will remove this module when all models using it are deprecated which may take time.

Defining a flag

absl flag definitions are similar to argparse, although they are defined on a global namespace.

For instance defining a string flag looks like:

```
from absl import flags
flags.DEFINE_string(
    name="my_flag",
    default="a_sensible_default",
    help="Here is what this flag does."
)
```

All three arguments are required, but default may be `None`. A common optional argument is short_name for defining abreviations. Certain `DEFINE_*` methods will have other required arguments. For instance `DEFINE_enum` requires the `enum_values` argument to be specified.

Key Flags

absl has the concept of a key flag. Any flag defined in `__main__` is considered a key flag by default. Key flags are displayed in `--help`, others only appear in `--helpfull`. In order to handle key flags that are defined outside the module in question, absl provides the `flags.adopt_module_key_flags()` method. This adds the key flags of a different module to one's own key flags. For example:

```
File: flag_source.py
---------------------------------------

from absl import flags
flags.DEFINE_string(name="my_flag", default="abc", help="a flag.")
```

```
File: my_module.py
---------------------------------------

from absl import app as absl_app
from absl import flags

import flag_source

flags.adopt_module_key_flags(flag_source)
```

```
def main(_):
  pass

absl_app.run(main, [__file__, "-h"]
```

when `my_module.py` is run it will show the help text for `my_flag`. Because not all flags defined in a file are equally important, `official/utils/flags/core.py` (generally imported as flags_core) provides an abstraction for handling key flag declaration in an easy way through the `register_key_flags_in_core()` function, which allows a module to make a single `adopt_key_flags(flags_core)` call when using the util flag declaration functions.

Validators

Often the constraints on a flag are complicated. absl provides the validator decorator to allow one to mark a function as a flag validation function. Suppose we want users to provide a flag which is a palindrome.

```
from absl import flags

flags.DEFINE_string(name="pal_flag", short_name="pf", default="", help="Give me a palindrome

@flags.validator("pal_flag")
def _check_pal(provided_pal_flag):
  return provided_pal_flag == provided_pal_flag[::-1]
```

Validators take the form that returning True (truthy) passes, and all others (False, None, exception) fail.

Testing

To test using absl, simply declare flags in the setupClass method of TensorFlow's TestCase.

```
from absl import flags
import tensorflow as tf

def define_flags():
  flags.DEFINE_string(name="test_flag", default="abc", help="an example flag")


class BaseTester(unittest.TestCase):

  @classmethod
  def setUpClass(cls):
    super(BaseTester, cls).setUpClass()
    define_flags()
```

```
def test_trivial(self):
  flags_core.parse_flags([__file__, "test_flag", "def"])
  self.AssertEqual(flags.FLAGS.test_flag, "def")
```