

packages_collection_part1

December 10, 2021

1 Content

Packages that are useful to know

- [argparse](#): provide arguments and flags to your application
- [json](#): Interact with lightweight data-interchange format JSON
- [tqdm](#): Add a Progress bar
- [re](#): handle regular expressions
- [subprocess](#): Spawn new processes

1.1 argparse

There are three possibilities to manipulate the behaviour of your application at runtime: 0. Change the code (not really a change at runtime nor user-friendly) 1. Execute your application with arguments on the command line 2. Read in configuration/initialisation files at runtime (not discussed in this workshop)

We will focus on approach to provide your application arguments via the command line and present three different packages; they are trading how easy they are to use or implement against their richness of features.

1.1.1 Using sys

By default to provide arguments to you python application you just add them on the command line:

```
$ ./main.py arg1 arg2 arg3
```

To access those arguments you can use the `sys` module:

```
[ ]: import sys

#fake terminal input

sys.argv=["application.py", "hallo welt", "5"]
print(f"You entered {len(sys.argv)} argument(s)")
for argument in sys.argv:
    print(f"{argument} (type: {type(argument)})")
```

This method is easy to use but has several drawbacks: 1. The first argument is always the application name, you must consider this if you want to access the arguments 2. All arguments are parsed

as strings, therefore you need to convert them 3. There is no association between arguments; if you want to check for a flag and parse its value you have to do it on your own

1.1.2 Using getopt

To help you parsing command line arguments there are several packages. One package for trivial parsing is `getopt`, which takes `sys.argv` and an option string.

```
[ ]: import sys, getopt
      from typing import Dict

      def parse_input(argv):
          parsed_args: Dict={}
          try:
              opts, args = getopt.getopt(argv[1:], "hab:c:", ["arg_b=", "arg_c="])
          except getopt.GetoptError:
              print(f"{sys.argv[0]} -a -b <argument> -c <argument>")
              return
          for opt, arg in opts:
              if opt == '-h':
                  print(f"{sys.argv[0]} -a -b <argument> -c <argument>")
                  return
              elif opt == "-a":
                  parsed_args[opt] = None
              elif opt in ("-b", "--arg_b"):
                  parsed_args[opt] = arg
              elif opt in ("-c", "--arg_c"):
                  parsed_args[opt] = arg

          print(f"Parsed arguments: {str(parsed_args)}")
          print(f"Omitted arguments: {str(args)}")
```

A few execution examples with faked terminal input:

```
[ ]: parse_input(["application.py"])
```

```
[ ]: parse_input(["application.py", "-d"])
```

```
[ ]: parse_input(["application.py", "hello"])
```

```
[ ]: parse_input(["application.py", "-a"])
```

```
[ ]: parse_input(["application.py", "hello", "-a", "world"])
```

```
[ ]: parse_input(["application.py", "-a", "-b", "foo", "--arg_c=bar"])
```

1.1.3 Using argparse

Setting up the argparse environment takes more lines of code but is rich on convenient features. A minimal working example looks like this:

```
[ ]: import argparse
      parser = argparse.ArgumentParser()
      parser.add_argument("foo")
      args = parser.parse_args(["Input A"])
      print(args.foo)
```

argparse does automatically provide a help text, especially if you also set description and help texts (try-except only for demonstration purpose)

```
[ ]: try:
      parser = argparse.ArgumentParser(description="This application is for
      ↪testing purpose")
      parser.add_argument("foo", help="First required argument")
      parser.add_argument("--bar", help="First optional argument")
      parser.add_argument("--baz", help="Second optional argument")
      parser.parse_args(["-h"])
    except:
      pass
```

```
[ ]: args = parser.parse_args(["Test", "--baz", "hallo"])
      print(f"args: {args}")
      print(f"Parsed arguments: {args.foo}, {args.baz}")
```

Arguments can be gathered in groups for improved overview

```
[ ]: parser = argparse.ArgumentParser(description="Test application with subgroup")
      parser.add_argument("FOO")
      group = parser.add_argument_group("Test group 1")
      group.add_argument("BAR", help="First parameter of subgroup")
      try:
        parser.parse_args(["-h"])
      except:
        pass
```

There are many optional arguments, which you can add while creating a parser object or add arguments - we will have a short look at a selection and use another format class, to show default values.

```
[ ]: import pathlib

      parser = argparse.ArgumentParser(description="More complex example",
      ↪epilog="Have a nice day!", formatter_class=argparse.
      ↪ArgumentDefaultsHelpFormatter)
      parser.add_argument("foo", type=int, nargs=2)
```

```

parser.add_argument("bar", type=float, nargs=3)
parser.add_argument("-b", "--baz", "---bazz", type=pathlib.Path,
    ↪default="hello/world.txt")
parser.add_argument("-c", type=pathlib.Path, default="hello/world.txt", help="A
    ↪help text is needed, if the default value shall be visible")
try:
    parser.parse_args(["-h"])
except:
    pass

```

```

[ ]: args=parser.parse_args(["1", "2", "3.0", "4.0", "5.0", "-b", "some/path/to/file.
    ↪txt"])
print(args)
print(f"Type FOO: {type(args.foo[0])}")
print(f"Type BAR: {type(args.bar[2])}")
print(f"Type baz: {type(args.baz)}")
print(f"Type c: {type(args.c)}")

```