16 May 2018

Parsing Command Line Arguments

Using the `argparse` library to manage `argv`

The name means "argument parser"

- parser: an algorithm/process that analyzes structure
- the term comes from linguistics, as in "parse a sentence" (break it into noun phrase, verb phrase, etc)

Use `argparse` in scripts that have a complicated set of command line options

Why Do We Need a Parser?

Many applications have several runtime options

Example: BLAST

Here are some of the arguments we can specify when we run BLAST (a program that searches a sequence database):

```
$ blastp --help
USAGE
  blastp [-h] [-help]
    [-import_search_strategy filename]
    [-export_search_strategy filename]
    [-task task_name] [-db database_name]
    [-dbsize num_letters] [-gilist filename]
    [-seqidlist filename]
    [-negative_gilist filename]
    [-entrez_query entrez_query]
    [-db_soft_mask filtering_algorithm]
    [-db_hard_mask filtering_algorithm]
    ...
```

The command line options tell BLAST the name of the file with the input sequence, the path to the database, similarity cutoffs, and tons of other search parameters

Example: `vsearch`

`vsearch` is one of the programs we use in the 16S analysis pipeline

```
$ vsearch --help
vsearch v1.10.0_osx_x86_64, 64.0GB RAM, 16 cores
https://github.com/torognes/vsearch

Usage: vsearch [OPTIONS]

General options
  --fasta_width INT           width of FASTA lines
  --help | --h                display help
  --log FILENAME              write messages to file
  --maxseqlength INT          maximum sequence length
  ...
```

There are literally hundreds of options:

$ vsearch --help | grep \-\- - | wc

```
  251   2047  18015
```

Even Simple Programs Could Use Some Help

BLAST and `vsearch` are admittedly extreme examples

But even if a script has only a few command line inputs it's easy to forget what they mean

- example: `bmi.py`
- when we run the program, does weight come first, or height?

```
$ python bmi.py 200 72
1.3

$ python bmi.py 72 200
27.1
```

A user who knows BMI values would quickly recognize the first output is nonsense, and infer that height should come before weight

But in other programs…. ?

Give Names to Inputs

I rewrote the BMI program so that height and weight need to be specified by name:

```
$ bmi.py --height 72 --weight 200
27.1
```

The might seem more complicated — users now have to type four things after the program name instead of two

But there's a benefit: the arguments can be specified in either order:

```
$ bmi.py --weight 200 --height 72
27.1
```

Option Names Are Part of `argv`

The shell doesn't distinguish between option names (things that start with dashes) and values (numbers)

Anything that is typed on the command line is collected and put in `argv`

```
$ argvdemo.py --weight 200 --height 72
There are 5 command line arguments:
  1  "--weight"
  2  "200"
  3  "--height"
  4  "72"
```

It's up to the program to look at the arguments and figure out what is an option name and to save the correct values

The Wrong Way to Check for Option Names

A program that checks for option names can get pretty messy…

```
# ** ugly version **  how NOT to write the program **

  from sys import argv

  if argv[1] == '--weight':
      weight = int(argv[2])
      if argv[3] == '--height':
          height = int(argv[4])
      else:
          print('unknown option:', argv[3])
          exit()
  elif argv[1] == '--height':
      height = int(argv[2])
      if argv[3] == '--weight':
          weight = int(argv[4])
          # etc
```

The `argparse` Library

The standard library named `argparse` will do all the hard work for us

Step 1: Create the Argument Parser

The parser is an **object** — create it by a function call:

```
parser = argparse.ArgumentParser()
```

Step 2: Tell the Parser About Your Command Line Options

Call a method to define each argument in your program:

```
parser.add_argument('--height', type=int)
parser.add_argument('--weight', type=int)
```

Step 3: Call the Parser

Call another method to invoke the parser, get back an object that contains all the command line options:

```
args = parser.parse_args()
bmi = 703 * args.weight / args.height ** 2
```

> *Notice how the parser did the type conversion for us — the height and weight values are automatically converted to integers*

BMI: The `argparse` Version

This is the complete text of my new version of the BMI program.

- it shows some additional information we can pass when creating the parser and calling `add_argument`

```
# Compute body mass index (BMI) given height
# in inches and weight in pounds

import argparse

parser = argparse.ArgumentParser(
        description="Compute body mass index",
)

parser.add_argument('--height', type=int,
   required=True, help='height, in inches')

parser.add_argument('--weight', type=int,
   required=True, help='weight, in pounds')

args = parser.parse_args()

bmi = 703 * args.weight / args.height ** 2

print(round(bmi,1))
```

Help Message

As a bonus, the argument parser automatically creates a “usage string” for us, and includes a `--help` option.

```
$ bmi.py --help
usage: bmi.py [-h] --height H --weight W

Compute body mass index

optional arguments:
  -h, --help  show this help message and exit
  --height H  height, in inches
  --weight W  weight, in pounds
```

More Options for `add_argument`

I wrote a new version of my GPA calculator to illustrate a few of the other features of the argument parser

- you'll want to use some of these to earn full credit on one of the programs in Unit 6

Positional Arguments

In the BMI program all inputs are specified by ***name,*** but sometimes we want to get values and refer to them by their ***position***

The new GPA program uses this technique to allow users to enter the letter grades:

```
parser.add_argument('grades', nargs='+')
```

- notice how there are no dashes before the argument name ('grades')
- the "nargs" (number of arguments) specification means there needs to be one or more values
- the parser will collect all of the values and put them in a Python list

The new `for` statement in the program is

```
for grade in args.grades:
```

Options for `add_argument` (cont'd)

Default Value

We can tell the argument parser what value to use for an option if the user does not specify it on the command line.

Example: the default precision for printing the GPA is one decimal point, but we can supply a different value with `--precision`

```
$ gpa.py A B B
3.3

$ gpa.py A B B --precision 3
3.333
```

Here is how the `--precision` option is specified:

```
parser.add_argument('--precision', type=int,
    default=1)
```

And this is how it is used:

```
gpa = round(total/n, args.precision)
```

Options for `add_argument` (cont'd)

Choices

A program might want to limit the set of values a user can enter.

Example: by default the new GPA program does not use + and - on letter grades, but the user can change that with `--plus_minus`

```
$ gpa.py A B- B
unknown letter grade: B-

$ gpa.py A B- B --plus_minus yes
3.2
```

This example shows how to make sure the only possible values for this option are the words "yes" and "no":

```
parser.add_argument('--plus_minus',
    choices=['yes','no'], default='no')
```

Options for `add_argument` (cont'd)

Booleans (aka "flags")

A better way to specify a yes/no or true/false choice is to define an option as a "flag"

If the user specifies `--allow_pnp` in the new program it will accept P and N as grades but not count them in the calculation

```
$ gpa.py A P B
unknown letter grade: P

$ gpa.py A P B --allow_pnp
3.5
```

Options for `add_argument` (cont'd)

This is how the new flag is defined:

```
parser.add_argument('--allow_pnp', action='store_true')
```

Here's how the new option is used:

```
  for grade in args.grades:
      if grade in points:
          ...
      elif args.allow_pnp and grade in ['P', 'N']:
          continue
      else:
          print('unknown letter grade:', grade)
```

An example using both new options:

```
$ gpa.py A P B- B --allow_pnp --plus_minus yes
3.2
```

The New GPA Program

The complete text of the new GPA program is in a file named `ap_gpa.py` on Canvas

The calls to `add_argument` all include a help option — the parser uses these strings when it builds the "usage" string

```
$ gpa.py --help
```

```
usage: gpa.py [-h] [--precision P]
    [--plus_minus {yes,no}]
    [--allow_pnp]  grades [grades ...]

Compute grade point average

positional arguments:
  grades                sequence of letter grades

optional arguments:
  -h, --help            show help message and exit
  --precision P         number of digits
  --plus_minus {yes,no}
                        allow + and - grades
  --allow_pnp           ignore P and N
```

Challenge

Rewrite the mortgage payment program (`pmt.py`) from Unit 3 so it uses `argparse` to get the values from the command line.