Common Patterns in Python

> "To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge." - Grace Hopper

Get positional command-line arguments

You can get the command-line arguments using `sys.argv` (argument vector), but it's annoying that the name of the Python program itself is in the first position (`sys.argv[0]`). To skip over this, take a slice of the argument vector starting at the second position (index `1`) which will succeed even if there are no arguments – you'll get an empty list, which is safe.

```
$ cat -n args.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
     7  num = len(args)
     8
     9  print('There are {} arg{}'.format(num, '' if num == 1 else 's'))
$ ./args.py
There are 0 args
$ ./args.py foo
There are 1 arg
$ ./args.py foo bar
There are 2 args
```

Put positional arguments into named variables

If you use `sys.argv[1]` and `sys.argv[2]` throughout your program, it degrades readability. It's better to copy the values into variables that have meaningful names like "file" or "num_lines".

```
$ cat -n name_args.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
```

```
     7
     8  if len(args) != 2:
     9      print('Usage: {} FILE NUM'.format(os.path.basename(sys.argv[0])))
    10      sys.exit(1)
    11
    12  file, num = args
    13
    14  file = args[0]
    15  num = args[1]
    16
    17  print('FILE is "{}", NUM is "{}"'.format(file, num))
$ ./name_args.py
Usage: name_args.py FILE NUM
$ ./name_args.py nobody.txt 10
FILE is "nobody.txt", NUM is "10"
```

Set defaults for optional arguments

```
$ cat -n default_arg.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
     7  num_args = len(args)
     8
     9  if not 1 <= num_args <= 2:
    10      print('Usage: {} FILE [NUM]'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  file = args[0]
    14  num = args[1] if num_args == 2 else 10
    15
    16  print('FILE is "{}", NUM is "{}"'.format(file, num))
$ ./default_arg.py
Usage: default_arg.py FILE [NUM]
$ ./default_arg.py nobody.txt
FILE is "nobody.txt", NUM is "10"
$ ./default_arg.py nobody.txt 5
FILE is "nobody.txt", NUM is "5"
```

Test argument is file and read

This program takes an argument, tests that it is a file, and then reads it. It's basically `cat`.

```
$ cat -n read_file.py
     1  #!/usr/bin/env python3
     2  """Read a file argument"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  filename = args[0]
    14
    15  if not os.path.isfile(filename):
    16      print('"{}" is not a file'.format(filename), file=sys.stderr)
    17      sys.exit(1)
    18
    19  for line in open(filename):
    20      print(line, end='')
$ ./read_file.py foo
"foo" is not a file
$ ./read_file.py nobody.txt
I'm Nobody! Who are you?
Are you - Nobody - too?
Then there's a pair of us!
Don't tell! they'd advertise - you know!

How dreary - to be - Somebody!
How public - like a Frog -
To tell one's name - the livelong June -
To an admiring Bog!

Emily Dickinson
```

Write data to a file

To write a file, you need to `open` some filename with a second argument of the "mode" where

- `r`: read (default)
- `w`: write
- `t`: text mode (default)
- `b`: binary

You can combine the flags so that `wt` means "write a text file" which is what is done here.

If you `open` a file for writing and the file already exists, it will be overwritten, so it may behoove you to check if the file exists first!

```
$ cat -n write_file.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) < 1:
    10      print('Usage: {} ARG1 [ARG2...]'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  outfile = 'out.txt'
    14  out_fh = open(outfile, 'wt')
    15
    16  for arg in args:
    17      out_fh.write(arg + '\n')
    18
    19  out_fh.close()
    20  print('Done, see "{}"'.format(outfile))
$ ./write_file.py foo bar baz
Done, see "out.txt"
$ cat out.txt
foo
bar
baz
```

Test if an argument is a directory and list the contents

```
$ cat -n list_dir.py
     1  #!/usr/bin/env python3
     2  """Show contents of directory argument"""
     3
     4  import os
```

```
     5  import sys
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} DIR'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  dirname = args[0]
    14
    15  if not os.path.isdir(dirname):
    16      print('"{}" is not a directory'.format(dirname), file=sys.stderr)
    17      sys.exit(1)
    18
    19  for entry in os.listdir(dirname):
    20      print(entry)
$ ./list_dir.py
Usage: list_dir.py DIR
$ ./list_dir.py .
list_dir.py
kmers.py
skip_loop.py
unpack_dict2.py
nobody.txt
name_args.py
create_dir.py
sort_dict_by_values.py
foo
args.py
sort_dict_by_keys.py
read_file.py
sort_dict_by_keys2.py
unpack_dict.py
codons.py
default_arg.py
```

Skip an iteration of a loop

Sometimes in a loop (`for` or `while`) you want to skip immediately to the top of the loop. You can use `continue` to do this. In this example, we skip the even-numbered lines by using the modulus `%` operator to find those line numbers which have a remainder of 0 after dividing by 2. We can use the `enumerate` function to provide both the array index and value of any list.

```
$ cat -n skip_loop.py
```

```
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  file = args[0]
    14
    15  if not os.path.isfile(file):
    16      print('"{}" is not a file'.format(file), file=sys.stderr)
    17      sys.exit(1)
    18
    19  for i, line in enumerate(open(file)):
    20      if (i + 1) % 2 == 0:
    21          continue
    22
    23      print(i + 1, line, end='')
$ ./skip_loop.py
Usage: skip_loop.py FILE
$ ./skip_loop.py nobody.txt
1 I'm Nobody! Who are you?
3 Then there's a pair of us!
5
7 How public - like a Frog -
9 To an admiring Bog!
11 Emily Dickinson
```

Create a directory if it does not exist

This program takes a directory name and looks to see if it already exists or needs to be created.

```
$ cat -n create_dir.py
     1  #!/usr/bin/env python3
     2  """Test for a directory and create if needed"""
     3
     4  import os
     5  import sys
     6
```

```
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} DIR'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  dirname = args[0]
    14
    15  if os.path.isdir(dirname):
    16      print('"{}" exists'.format(dirname))
    17  else:
    18      print('Creating "{}"'.format(dirname))
    19      os.makedirs(dirname)
$ ./create_dir.py
Usage: create_dir.py DIR
$ ./create_dir.py foo
Creating "foo"
$ ./create_dir.py foo
"foo" exists
```

Unpack a dictionary's key/values pairs

The `.items()` method on a dictionary will return a list of tuples:

```
$ cat -n unpack_dict.py
     1  #!/usr/bin/env python3
     2  """Unpack dict"""
     3
     4  import os
     5  import sys
     6
     7  albums = {
     8      "2112": 1976,
     9      "A Farewell To Kings": 1977,
    10      "All the World's a Stage": 1976,
    11      "Caress of Steel": 1975,
    12      "Exit, Stage Left": 1981,
    13      "Fly By Night": 1975,
    14      "Grace Under Pressure": 1984,
    15      "Hemispheres": 1978,
    16      "Hold Your Fire": 1987,
    17      "Moving Pictures": 1981,
    18      "Permanent Waves": 1980,
    19      "Power Windows": 1985,
```

```
    20      "Signals": 1982,
    21  }
    22
    23  for tup in albums.items():
    24      album = tup[0]
    25      year = tup[1]
    26      print('{:4} {}'.format(year, album))
$ ./unpack_dict.py
1976 2112
1977 A Farewell To Kings
1976 All the World's a Stage
1975 Caress of Steel
1981 Exit, Stage Left
1975 Fly By Night
1984 Grace Under Pressure
1978 Hemispheres
1987 Hold Your Fire
1981 Moving Pictures
1980 Permanent Waves
1985 Power Windows
1982 Signals
```

But the `for` loop could unpack the tuple directly. Compare line 23 in the above and below programs.

```
$ cat -n unpack_dict2.py
     1  #!/usr/bin/env python3
     2  """Unpack dict"""
     3
     4  import os
     5  import sys
     6
     7  albums = {
     8      "2112": 1976,
     9      "A Farewell To Kings": 1977,
    10      "All the World's a Stage": 1976,
    11      "Caress of Steel": 1975,
    12      "Exit, Stage Left": 1981,
    13      "Fly By Night": 1975,
    14      "Grace Under Pressure": 1984,
    15      "Hemispheres": 1978,
    16      "Hold Your Fire": 1987,
    17      "Moving Pictures": 1981,
    18      "Permanent Waves": 1980,
    19      "Power Windows": 1985,
    20      "Signals": 1982,
    21  }
```

```
    22
    23  for album, year in albums.items():
    24      print('{:4} {}'.format(year, album))
$ ./unpack_dict2.py
1976 2112
1977 A Farewell To Kings
1976 All the World's a Stage
1975 Caress of Steel
1981 Exit, Stage Left
1975 Fly By Night
1984 Grace Under Pressure
1978 Hemispheres
1987 Hold Your Fire
1981 Moving Pictures
1980 Permanent Waves
1985 Power Windows
1982 Signals
```

Sort a dictionary by keys

To sort a dictionary by the keys, you have to understand that the `.sort()` method of an list mutates the list *in-place*. We get the keys of a dictionary with the `.keys()` method which does not support the `.sort()` method:

```
>>> d = dict(foo=1, bar=2)
>>> d.keys()
dict_keys(['foo', 'bar'])
>>> d.keys().sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict_keys' object has no attribute 'sort'
```

We could copy the keys into a list to sort like so:

```
>>> k = list(d.keys())
>>> k
['foo', 'bar']
>>> k.sort()
>>> k
['bar', 'foo']
```

Or we can use the `sorted()` function that accepts a list and *returns a sorted list*:

```
>>> d.keys()
dict_keys(['foo', 'bar'])
>>> sorted(d.keys())
```

```
['bar', 'foo']
```

Either way, once we have the sorted keys, we can get the associated values:

```
$ cat -n sort_dict_by_keys.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  albums = {
     7      "2112": 1976,
     8      "A Farewell To Kings": 1977,
     9      "All the World's a Stage": 1976,
    10      "Caress of Steel": 1975,
    11      "Exit, Stage Left": 1981,
    12      "Fly By Night": 1975,
    13      "Grace Under Pressure": 1984,
    14      "Hemispheres": 1978,
    15      "Hold Your Fire": 1987,
    16      "Moving Pictures": 1981,
    17      "Permanent Waves": 1980,
    18      "Power Windows": 1985,
    19      "Signals": 1982,
    20  }
    21
    22  for album in sorted(albums.keys()):
    23      print('{:25} {}'.format(album, albums[album]))
$ ./sort_dict_by_keys.py
2112                      1976
A Farewell To Kings       1977
All the World's a Stage   1976
Caress of Steel           1975
Exit, Stage Left          1981
Fly By Night              1975
Grace Under Pressure      1984
Hemispheres               1978
Hold Your Fire            1987
Moving Pictures           1981
Permanent Waves           1980
Power Windows             1985
Signals                   1982
```

Or we could unpack the tuples directly like above:

```
$ cat -n sort_dict_by_keys2.py
     1  #!/usr/bin/env python3
     2
```

```
     3  import os
     4  import sys
     5
     6  albums = {
     7      "2112": 1976,
     8      "A Farewell To Kings": 1977,
     9      "All the World's a Stage": 1976,
    10      "Caress of Steel": 1975,
    11      "Exit, Stage Left": 1981,
    12      "Fly By Night": 1975,
    13      "Grace Under Pressure": 1984,
    14      "Hemispheres": 1978,
    15      "Hold Your Fire": 1987,
    16      "Moving Pictures": 1981,
    17      "Permanent Waves": 1980,
    18      "Power Windows": 1985,
    19      "Signals": 1982,
    20  }
    21
    22  for album, year in sorted(albums.items()):
    23      print('{:25} {}'.format(album, year))
$ ./sort_dict_by_keys2.py
2112                      1976
A Farewell To Kings       1977
All the World's a Stage   1976
Caress of Steel           1975
Exit, Stage Left          1981
Fly By Night              1975
Grace Under Pressure      1984
Hemispheres               1978
Hold Your Fire            1987
Moving Pictures           1981
Permanent Waves           1980
Power Windows             1985
Signals                   1982
```

Sort a dictionary by values

To sort a dictionary by the values rather than the keys, we need to reverse the tuples which is what happens on line 24. Notice that in years when two albums were released, the `sorted` first sorts by the first tuple member (the year) and then the second (album name):

```
$ cat -n sort_dict_by_values.py
     1  #!/usr/bin/env python3
```

```
 2
 3  import os
 4  import sys
 5
 6  albums = {
 7      "2112": 1976,
 8      "A Farewell To Kings": 1977,
 9      "All the World's a Stage": 1976,
10      "Caress of Steel": 1975,
11      "Exit, Stage Left": 1981,
12      "Fly By Night": 1975,
13      "Grace Under Pressure": 1984,
14      "Hemispheres": 1978,
15      "Hold Your Fire": 1987,
16      "Moving Pictures": 1981,
17      "Permanent Waves": 1980,
18      "Power Windows": 1985,
19      "Signals": 1982,
20  }
21
22  # Create a list of (value, key) tuples
23  # sorted in descending order by the values
24  pairs = sorted([(x[1], x[0]) for x in albums.items()])
25
26  for year, album in pairs:
27      print('{} {}'.format(year, album))
$ ./sort_dict_by_values.py
1975 Caress of Steel
1975 Fly By Night
1976 2112
1976 All the World's a Stage
1977 A Farewell To Kings
1978 Hemispheres
1980 Permanent Waves
1981 Exit, Stage Left
1981 Moving Pictures
1982 Signals
1984 Grace Under Pressure
1985 Power Windows
1987 Hold Your Fire
```

Extract codons from DNA

This example assumes a codon length (`k`) of 3 and uses a handy third argument to `range` that indicates the distance to skip in each iteration. The goal is to start at position 0, then jump to position 3, then 6, etc., to extract all the codons. Imagine how you could expand this to get all the codons in all the frames (this one starts at "1" which is really "0" in the string):

```
$ cat -n codons.py
     1  #!/usr/bin/env python3
     2  """Extract codons from DNA"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8  num_args = len(args)
     9
    10  if not 1 <= num_args <= 2:
    11      print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    12      sys.exit(1)
    13
    14  string = args[0]
    15  k = 3
    16  n = len(string) - k + 1
    17
    18  for i in range(0, n, k):
    19      print(string[i:i+k])
$ ./codons.py
Usage: codons.py DNA
$ ./codons.py AAACCCGGGTTT
AAA
CCC
GGG
TTT
```

Extract k-mers from a string

K-mers are k-length contiguous sub-sequences from a string. They are similar to codons (which are 3-mers), but we tend to move across the string by one character than than the codon length (3). Notice this script guards against a 2nd argument that should be a number but is not:

```
$ cat -n kmers.py
     1  #!/usr/bin/env python3
```

```
     2  """Extract k-mers from string"""
     3
     4  import os
     5  import sys
     6
     7  args = sys.argv[1:]
     8  num_args = len(args)
     9
    10  if not 1 <= num_args <= 2:
    11      print('Usage: {} STR [K]'.format(os.path.basename(sys.argv[0])))
    12      sys.exit(1)
    13
    14  string = args[0]
    15  k = args[1] if num_args == 2 else '3'
    16
    17  # Guard against a string like "foo"
    18  if not k.isdigit():
    19      print('k "{}" is not a digit'.format(k))
    20      sys.exit(1)
    21
    22  # Safe to convert now
    23  k = int(k)
    24
    25  if len(string) < k:
    26      print('There are no {}-length substrings in "{}"'.format(k, string))
    27  else:
    28      n = len(string) - k + 1
    29      for i in range(0, n):
    30          print(string[i:i+k])
$ ./kmers.py
Usage: kmers.py STR [K]
$ ./kmers.py foobar 10
There are no 10-length substrings in "foobar"
$ ./kmers.py AAACCCGGGTTT 3
AAA
AAC
ACC
CCC
CCG
CGG
GGG
GGT
GTT
TTT
```

Make All Items in a List Uppercase

If you need to check all the strings in a list in a case-insensitive fashion, one way would be to upper- or lower-case all the strings. A very Pythonic way is to use a list comprehension, but we can also borrow an idea from the purely functional programming world where we use a "higher-order function," which is a function that takes one or more other functions as arguments. In this case, the `map` function expects as it's first argument some other function, here `str.upper`. Notice it's not `str.upper()` with parens! That is the syntax for **calling** the `str.upper` function. We want to pass **the function itself**, so we leave off the parens. The function is applied to each item in the list and returns a new list. The original list remains unchanged.

```
>>> a = ['foo', 'bar', 'baz']
>>> [s.upper() for s in a]
['FOO', 'BAR', 'BAZ']
>>> list(map(str.upper, a))
['FOO', 'BAR', 'BAZ']
>>> a
['foo', 'bar', 'baz']
```

Another way to write the `map` is to use a `lambda` expression which is just a very short, anonymouse (unnamed) function:

```
>>> list(map(lambda s: s.upper(), a))
['FOO', 'BAR', 'BAZ']
```

Here is the code in action:

```
$ cat -n upper_list.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import sys
     5
     6  args = sys.argv[1:]
     7
     8  if len(args) < 1:
     9      print('Usage: {} ARG [ARG...]'.format(os.path.basename(sys.argv[0])))
    10      sys.exit(1)
    11
    12  print('List comprehension')
    13  print(', '.join([x.upper() for x in args]))
    14
    15  print('Map')
    16  print(', '.join(map(str.upper, args)))
$ ./upper_list.py foo bar baz
List comprehension
```

```
FOO, BAR, BAZ
Map
FOO, BAR, BAZ
```