

Writing Simple Games in Python

Games are a terrific way to learn. If you take something simple you know well, you have all the information you need to complete it. Something simple like tic-tac-toe – you know you need a board, some way for the user to select a cell, you need to keep track of who’s playing (X or O), when they’ve made a bad move, and when someone has won. Games often need random values, interact with the user, employ infinite loops – in short, they are fascinating and fun to program and play.

Twelve Days of Christmas

Here is an implementation of the “Twelve Days of Christmas” song. It uses two loops to count up from 1 for each step and then a countdown from each step back to 1. Notice I use integers as the keys to the dictionaries.

```
$ cat -n twelve_days.py
 1  #!/usr/bin/env python3
 2  """
 3  Author : kycklar
 4  Date   : 2019-03-19
 5  Purpose: Twelve Days of Christmas
 6  """
 7
 8  import argparse
 9  import sys
10
11
12  # -----
13  def get_args():
14      """get command-line arguments"""
15      parser = argparse.ArgumentParser(
16          description='Twelve Days of Christmas',
17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
18
19      parser.add_argument(
20          '-o',
21          '--outfile',
22          help='Outfile (STDOUT)',
23          metavar='str',
24          type=str,
25          default='')
26
27      parser.add_argument(
```

```

28         '-n',
29         '--number_days',
30         help='Number of days to sing',
31         metavar='int',
32         type=int,
33         default=12)
34
35     return parser.parse_args()
36
37
38     # -----
39     def warn(msg):
40         """Print a message to STDERR"""
41         print(msg, file=sys.stderr)
42
43
44     # -----
45     def die(msg='Something bad happened'):
46         """warn() and exit with error"""
47         warn(msg)
48         sys.exit(1)
49
50
51     # -----
52     def main():
53         """Make a jazz noise here"""
54         args = get_args()
55         out_file = args.outfile
56         num_days = args.number_days
57
58         out_fh = open(out_file, 'wt') if out_file else sys.stdout
59
60         days = {
61             12: 'Twelve drummers drumming',
62             11: 'Eleven pipers piping',
63             10: 'Ten lords a leaping',
64             9: 'Nine ladies dancing',
65             8: 'Eight maids a milking',
66             7: 'Seven swans a swimming',
67             6: 'Six geese a laying',
68             5: 'Five gold rings',
69             4: 'Four calling birds',
70             3: 'Three French hens',
71             2: 'Two turtle doves',
72             1: 'a partridge in a pear tree',
73         }

```

```

74
75     cardinal = {
76         12: 'twelfth',
77         11: 'eleven',
78         10: 'tenth',
79         9: 'ninth',
80         8: 'eighth',
81         7: 'seventh',
82         6: 'sixth',
83         5: 'fifth',
84         4: 'fourth',
85         3: 'third',
86         2: 'second',
87         1: 'first',
88     }
89
90     if not num_days in days:
91         die('Cannot sing "{}" days'.format(num_days))
92
93     def ucfirst(s):
94         return s[0].upper() + s[1:]
95
96     for i in range(1, num_days + 1):
97         first = 'On the {} day of Christmas,\nMy true love gave to me,'
98         out_fh.write(first.format(cardinal[i]) + '\n')
99         for j in reversed(range(1, i + 1)):
100             if j == 1:
101                 if i == 1:
102                     out_fh.write('{}.\n'.format(ucfirst(days[j])))
103                 else:
104                     out_fh.write('And {}.\n'.format(days[j]))
105             else:
106                 out_fh.write('{}.\n'.format(days[j]))
107
108         if i < max(days.keys()):
109             out_fh.write('\n')
110
111
112 # -----
113 if __name__ == '__main__':
114     main()

```

Guessing Game

Let's write a simple program where the user has to guess a random number.
First, let's sketch out some pseudo-code:

```
Establish the range of numbers allowed and how many times the player can guess
Pick a random number in that range
Start a loop
```

```
    Ask the user for a guess
```

```
    Quit if the user asks
```

```
    Make sure the guess is a number and in the allowed range
```

```
    If the number was correctly guessed, stop and tell the user they won
```

```
    Let the user know if the number is high or low
```

```
    If the user has guessed too many times, stop and tell the user they lost (also insult th
```

Here's how it looks being played. Note that I use a binary search where I divide the search space in half on each guess. I can usually guess the number correctly in 5 guesses when the range is 1-50.

```
$ ./guess.py
Guess a number between 1 and 50 (q to quit): 25
"25" is too low.
Guess a number between 1 and 50 (q to quit): 37
"37" is too low.
Guess a number between 1 and 50 (q to quit): 45
"45" is too high.
Guess a number between 1 and 50 (q to quit): 40
"40" is too high.
Guess a number between 1 and 50 (q to quit): 38
"38" is correct. You win!
$ ./guess.py -x 100
Guess a number between 1 and 100 (q to quit): 50
"50" is too low.
Guess a number between 1 and 100 (q to quit): 75
"75" is too high.
Guess a number between 1 and 100 (q to quit): 62
"62" is too high.
Guess a number between 1 and 100 (q to quit): 55
"55" is too low.
Guess a number between 1 and 100 (q to quit): 58
"58" is too low.
Too many guesses, loser! The number was "59."
$ ./guess.py
Guess a number between 1 and 50 (q to quit): quit
Now you will never know the answer.
```

As usual, we'll start with `new_py.py`, and I'll use `argparse` to get the min/max

range with defaults of 1/50 and set the number of guesses to 5:

```
$ cat -n guess.py
 1  #!/usr/bin/env python3
 2  """
 3  Author:  Ken Youens-Clark <kyclark@gmail.com>
 4  Purpose: Guess-the-number game
 5  """
 6
 7  import argparse
 8  import random
 9  import re
10  import sys
11
12
13  # -----
14  def get_args():
15      """get args"""
16      parser = argparse.ArgumentParser(
17          description='Guessing game',
18          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
19
20      parser.add_argument(
21          '-m',
22          '--min',
23          help='Minimum value',
24          metavar='int',
25          type=int,
26          default=1)
27
28      parser.add_argument(
29          '-x',
30          '--max',
31          help='Maximum value',
32          metavar='int',
33          type=int,
34          default=50)
35
36      parser.add_argument(
37          '-g',
38          '--guesses',
39          help='Number of guesses',
40          metavar='int',
41          type=int,
42          default=5)
43
```

```

44     return parser.parse_args()
45
46
47 # -----
48 def warn(msg):
49     """Print a message to STDERR"""
50     print(msg, file=sys.stderr)
51
52
53 # -----
54 def die(msg='Something bad happened'):
55     """warn() and exit with error"""
56     warn(msg)
57     sys.exit(1)
58
59
60 # -----
61 def main():
62     """main"""
63     args = get_args()
64     low = args.min
65     high = args.max
66     guesses_allowed = args.guesses
67     secret = random.randint(low, high)
68
69     if low < 1:
70         die('--min "{}" cannot be lower than 1'.format(low))
71
72     if guesses_allowed < 1:
73         die('--guesses "{}" cannot be lower than 1'.format(high))
74
75     if low > high:
76         die('--min "{}" is higher than --max "{}"'.format(low, high))
77
78     prompt = 'Guess a number between {} and {} (q to quit): '.format(low, high)
79     num_guesses = 0
80
81     while True:
82         guess = input(prompt)
83         num_guesses += 1
84
85         if re.match('q(uit)?', guess):
86             die('Now you will never know the answer.')
87
88         if not guess.isdigit():
89             warn("{} is not a number".format(guess))

```

```

90             continue
91
92         num = int(guess)
93
94         if not low <= num <= high:
95             print('Number "{}" is not in the allowed range'.format(num))
96         elif num == secret:
97             print("{} is correct. You win!".format(num))
98             break
99         else:
100             print("{} is too {}.format(num, 'low'
101                                     if num < secret else 'high'))
102
103         if num_guesses >= guesses_allowed:
104             die('Too many guesses, loser! The number was "{}.format(secret))
105
106
107     # -----
108     if __name__ == '__main__':
109         main()

```

The `get_args` function will ensure we get some values for the range and number of guesses, but we should always assume garbage from the user, so we have to check them. To get a random number in our given range, we use the `random` module's `randint` function:

```
secret = random.randint(low, high)
```

If you ever need to “flip a coin” in your code, you can do this:

```

>>> import random
>>> random.randint(0,1)
1
>>> random.randint(0,1)
0
>>> random.randint(0,1)
0

```

The meat of the program will be an infinite loop where we keep asking the user:

```
prompt = 'Guess a number between {} and {} (q to quit): '.format(low, high)
```

Before we enter that loop, we'll need a variable to keep track of the number of guesses the user has made. This is the lone piece of “state” we need to track. Other games can have many pieces of information you need to track.

```
num_guesses = 0
```

The beginning of the play loop looks like this:

```
while True:
```

```

guess = input('[{}] {}'.format(num_guesses, prompt))
num_guesses += 1

```

Here I want the user to know how many guesses they’ve made so far. We want to give them a way out, so they can enter “q” (or “quit”) to quit. I chose to use the `re` module for regular expressions so I can identify a string that is either “q” or “quit”. The bit in `()?` is considered optional because the parens group it and the question mark makes it optional:

```

if re.match('q(uit)?', guess.lower()):
    die('Now you will never know the answer.')

```

The input from the user will be a string, and we are going to need to convert it to an integer to see if it is the secret number. Before we do that, we must check that it is a digit. We can use the `isdigit` method that all strings have. Look at `help(str)` in your Python REPL to see other useful methods like `isalnum`, `isalpha`, `islower`, etc.:

```

if not guess.isdigit():
    warn("{} is not a number".format(guess))
    continue

```

If it’s not a digit, we `continue` to go to the next iteration of the loop. If we move ahead, then it’s OK to convert the guess by using the `int` method to coerce the string the user typed into an integer value:

```

>>> int('8')
8
>>> type(int('8'))
<type 'int'>

```

There’s an alternate way to handle the conversion of the guess using a `try/except` block (see `guess-try.py`) where the call to `int` is wrapped in a `try` block that has an `except` block that catches the exception that Python throws when it fails. Cf <https://docs.python.org/3/tutorial/errors.html>:

```

num = 0
try:
    num = int(guess)
except:
    warn("{} is not an integer".format(guess))
    continue

```

Now we need to determine if the user has guessed too many times, if the number is too high or low, or if they’ve won the game. Lastly we see if the user has exceeded the maximum number of guesses:

```

if not low <= num <= high:
    print('Number "{}" is not in the allowed range'.format(num))
elif num == secret:
    print("{} is correct. You win!".format(num))

```



```

        break
    else:
        print("{} is too {}".format(num, 'low'
                                     if num < secret else 'high'))

if num_guesses >= guesses_allowed:
    die('Too many guesses, loser! The number was {}'.format(secret))

```

Hangman

Here is an implementation of the game “Hangman” that uses dictionaries to maintain the “state” of the program – that is, all the information needed for each round of play such as the word being guessed, how many misses the user has made, which letters have been guessed, etc. The program uses the `argparse` module to gather options from the user while providing default values so that nothing needs to be provided. The `main` function is used just to gather the parameters and then run the `play` function which recursively calls itself, each time passing in the new “state” of the program. Inside `play`, we use the `get` method of `dict` to safely ask for keys that may not exist and use defaults. When the user finishes or quits, `play` will simply call `sys.exit` to stop.

Here is what it looks like being played:

```

$ ./hangman.py
- - - - - (Misses: 0)
Your guess? ("?" for hint, "!" to quit) a
_ a _ _ _ a _ _ (Misses: 0)
Your guess? ("?" for hint, "!" to quit) e
e a _ _ _ a _ _ (Misses: 0)
Your guess? ("?" for hint, "!" to quit) i
e a _ _ _ a _ _ (Misses: 1)
Your guess? ("?" for hint, "!" to quit) o
e a _ _ _ a _ _ (Misses: 2)
Your guess? ("?" for hint, "!" to quit) u
e a _ _ _ a _ _ (Misses: 3)
Your guess? ("?" for hint, "!" to quit) ?
e a _ _ h _ a _ _ (Misses: 4)
Your guess? ("?" for hint, "!" to quit) t
e a _ t h _ a _ _ (Misses: 4)
Your guess? ("?" for hint, "!" to quit) r
e a r t h _ a r _ (Misses: 4)
Your guess? ("?" for hint, "!" to quit) w
e a r t h w a r _ (Misses: 4)
Your guess? ("?" for hint, "!" to quit) d
You win. You guessed "earthward" with "4" misses!

```

Here is the code:

```
$ cat -n hangman.py
 1  #!/usr/bin/env python3
 2  """
 3  Author:  Ken Youens-Clark <kyclark@gmail.com>
 4  Purpose: Hangman game
 5  """
 6
 7  import argparse
 8  import os
 9  import random
10  import re
11  import sys
12
13
14  # -----
15  def get_args():
16      """parse arguments"""
17      parser = argparse.ArgumentParser(
18          description='Hangman',
19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
20
21      parser.add_argument(
22          '-l', '--maxlen', help='Max word length', type=int, default=10)
23
24      parser.add_argument(
25          '-n', '--minlen', help='Min word length', type=int, default=5)
26
27      parser.add_argument(
28          '-m', '--misses', help='Max number of misses', type=int, default=10)
29
30      parser.add_argument(
31          '-w',
32          '--wordlist',
33          help='Word list',
34          type=str,
35          default='/usr/share/dict/words')
36
37      return parser.parse_args()
38
39  # -----
40  def bail(msg):
41      """Print a message to STDOUT and quit with no error"""
42      print(msg)
43      sys.exit(0)
```

```

44
45 # -----
46 def warn(msg):
47     """Print a message to STDERR"""
48     print(msg, file=sys.stderr)
49
50
51 # -----
52 def die(msg='Something bad happened'):
53     """warn() and exit with error"""
54     warn(msg)
55     sys.exit(1)
56
57 # -----
58 def main():
59     """main"""
60     args = get_args()
61     max_len = args.maxlen
62     min_len = args.minlen
63     max_misses = args.misses
64     wordlist = args.wordlist
65
66     if not os.path.isfile(wordlist):
67         die('--wordlist "{}" is not a file.'.format(wordlist))
68
69     if min_len < 1:
70         die('--minlen must be positive')
71
72     if not 3 <= max_len <= 20:
73         die('--maxlen should be between 3 and 20')
74
75     if min_len > max_len:
76         die('--minlen ({}) is greater than --maxlen ({})'.format(
77             min_len, max_len))
78
79     good_word = re.compile('^([a-z]{' + str(min_len) + ',' + str(max_len) + '})$')
80     words = [w for w in open(wordlist).read().split() if good_word.match(w)]
81     word = random.choice(words)
82     play({'word': word, 'max_misses': max_misses})
83
84
85 # -----
86 def play(state):
87     """Loop to play the game"""
88     word = state.get('word') or ''
89

```

```

90     if not word: die('No word!')
91
92     guessed = state.get('guessed') or list('_' * len(word))
93     prev_guesses = state.get('prev_guesses') or set()
94     num_misses = state.get('num_misses') or 0
95     max_misses = state.get('max_misses') or 0
96
97     if ''.join(guessed) == word:
98         msg = 'You win. You guessed "{}" with "{}" miss{}!'
99         bail(msg.format(word, num_misses, '' if num_misses == 1 else 'es'))
100
101     if num_misses >= max_misses:
102         bail('You lose, loser! The word was "{}".'.format(word))
103
104     print('{} (Misses: {})'.format(' '.join(guessed), num_misses))
105     new_guess = input('Your guess? ("?" for hint, "!" to quit) ').lower()
106
107     if new_guess == '!':
108         bail('Better luck next time, loser.')
109     elif new_guess == '?':
110         new_guess = random.choice([x for x in word if x not in guessed])
111         num_misses += 1
112
113     if not re.match('[a-z]$', new_guess):
114         print('"{}" is not a letter'.format(new_guess))
115         num_misses += 1
116     elif new_guess in prev_guesses:
117         print('You already guessed that')
118     elif new_guess in word:
119         prev_guesses.add(new_guess)
120         last_pos = 0
121         while True:
122             pos = word.find(new_guess, last_pos)
123             if pos < 0:
124                 break
125             elif pos >= 0:
126                 guessed[pos] = new_guess
127                 last_pos = pos + 1
128     else:
129         num_misses += 1
130
131     play({
132         'word': word,
133         'guessed': guessed,
134         'num_misses': num_misses,
135         'prev_guesses': prev_guesses,

```

```

136         'max_misses': max_misses
137     })
138
139
140 # -----
141 if __name__ == '__main__':
142     main()

```

NB: I would mention that my approach to recursively calling the `play` function with a dictionary for state rather than creating an infinite `while` loop was influenced by my experience programming in the Elm language.

Some notes on code:

I don't want to guess at words that are too short or too long, so we set up `min_len` and `max_len` variables and then use those to build a regular expression that describes a string that is composed of alphabet characters in that range of length:

```
regex = re.compile('^ [a-z]{ ' + str(min_len) + ', ' + str(max_len) + ' }$')
```

To visualize this, pretend we set 5 and 10 for lower and upper bounds:

```

>>> min_len = 5
>>> max_len = 10
>>> '^ [a-z]{ ' + str(min_len) + ', ' + str(max_len) + ' }$'
'^ [a-z]{5,10}$'

```

Now perhaps you can see the regex we've created?

```

1 2      3      4
^ [a-z] {5,10} $

```

1. The beginning of the string
2. The character class composed of the letters from “a” to “z”
3. A length from 5 to 10
4. The end of the string

I chose to use `re.compile` to turn this into a variable containing the regex so I can use it:

```

>>> good_word = re.compile('^ [a-z]{ ' + str(min_len) + ', ' + str(max_len) + ' }$')
>>> good_word.match('foo')
>>> good_word.match('foobar')
<_sre.SRE_Match object at 0x103f42ed0>

```

There's a lot packed into this line:

```
words = [w for w in open(wordlist).read().split() if good_word.match(w)]
```

So I open the words file and read it and then immediately `split` into words. Then I use a “list comprehension” to set up a little `for` loop over each word to take it if the word matches the regex. I could have written it more verbosely,

but this way is quite succinct and correct in a way that is actually harder to get right in a longer version like so:

```
words = []
for line in open(wordlist):
    for word in line.split():
        if good_word.match(word):
            words.append(word)
```

Similar to the guessing game, we need to randomly choose from our `words` which the `random.choice` function does exactly:

```
>>> import random
>>> random.choice(['foo', 'bar', 'baz'])
'bar'
>>> random.choice(['foo', 'bar', 'baz'])
'foo'
```

With that, we can launch into the `play` with a minimal state:

```
play({'word': word, 'max_misses': max_misses})
```

The `play` function is defined receiving a single `state` variable which is expected to be a dictionary. I could have passed in each part of the state individually as named variables, but this way seems cleaner to me:

```
def play(state):
```

The first time through `play`, there will be no previous guesses, so I use `dict.get` to ask for this so my code won't blow up. If nothing is available, I create a new string by multiplying the `_` (underscore) character by the length of the word where the underscore will indicate to the user where a letter has not been guessed. Since I want to store this as a list and not a string, I use `list` to convert the string:

```
guessed = state.get('guessed') or list('_' * len(word))
```

I'd like to keep track of all the letters the user has previously guessed so that I can tell them when they guess the same letter twice. The best data structure for this is a `set` which is essentially a dictionary with values of `1` – we only care if a key is present or absent so the value is irrelevant.

```
prev_guesses = state.get('prev_guesses') or set()
```

First I need to see if the user has guessed the correct word:

```
if ''.join(guessed) == word:
    msg = 'You win. You guessed "{}" with "{}" miss{}!'
    bail(msg.format(word, num_misses, '' if num_misses == 1 else 'es'))
```

The `bail` function is one I wrote just for this program as there are several places where I needed to `print` a message and `exit` *without* an error code.

To get a new guess from the user, I use `input` and chain it to the `lower` method of the returned string to lowercase the value:

```
new_guess = input('Your guess? ("?" for hint, "!" to quit) ').lower()
```

Because `q` is a valid input from the user, I can't use it to `quit` so I decided to use `!`. I also wanted to show mercy by allowing hints with the `?`. I this is implemented using another list comprehension to find all the letters in `word` that are *not* in the `guessed` set and then randomly select from that list:

```
if new_guess == '!':
    bail('Better luck next time, loser.')
elif new_guess == '?':
    new_guess = random.choice([x for x in word if x not in guessed])
    num_misses += 1
```

I use a regex similar to the `good_word` to see if we have exactly one character that is a letter:

```
>>> re.match('^[a-z]$', 'x')
<_sre.SRE_Match object at 0x103f76850>
>>> re.match('^[a-z]$', '4')
>>> re.match('^[a-z]$', '>')
```

Sets make it easy to check for membership:

```
elif new_guess in prev_guesses:
    print('You already guessed that')
```

Then we check if the guess is a character in the word; if so, add it to our previous guesses:

```
elif new_guess in word:
    prev_guesses.add(new_guess)
```

This next bit is tricky. We need to `find` the position(s) of the character in the word. Since the character may occur more than once, we need to keep track of the last position where we found it and use that as the second optional argument to `find`. E.g., in “foo” the “o” occurs twice. Keeping in mind zero-based counting:

```
>>> zip(range(3), 'foo')
[(0, 'f'), (1, 'o'), (2, 'o')]
```

`'foo'.find('o')` will always return 1 unless we tell it to start looking after that position:

```
>>> 'foo'.find('o')
1
>>> 'foo'.find('o')
1
>>> 'foo'.find('o', 2)
2
```

So we set up a variable `last_pos` to set to *after* whatever `find` returns. We need that in order to turn the underscore in `guessed` into the actual letter. Note that `find` will return `-1` to indicate no matches.

```
last_pos = 0
while True:
    pos = word.find(new_guess, last_pos)
    if pos < 0:
        break
    elif pos >= 0:
        guessed[pos] = new_guess
        last_pos = pos + 1
```

Finally we call `play` again but with a new state:

```
play({
    'word': word,
    'guessed': guessed,
    'num_misses': num_misses,
    'prev_guesses': prev_guesses,
    'max_misses': max_misses
})
```

This is an example of a recursive algorithm (<https://readtheplaque.com/plaque/the-toronto-recursive-history-project>), and they only work if you first handle the “base case.” For Hangman, that is where the user guesses the word or exceeds the number of guesses, both of which will **bail** on the program; otherwise, the program continues to the next iteration.