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.

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
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Guess-the-number game
"""

import argparse
import random
import re
import sys


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Guessing game',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        '-m',
        '--min',
        help='Minimum value',
        metavar='int',
        type=int,
        default=1)

    parser.add_argument(
        '-x',
        '--max',
        help='Maximum value',
        metavar='int',
        type=int,
        default=50)

    parser.add_argument(
        '-g',
        '--guesses',
        help='Number of guesses',
        metavar='int',
        type=int,
        default=5)

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    low = args.min
    high = args.max
    guesses_allowed = args.guesses
    secret = random.randint(low, high)

    if low < 1:
        die('--min "{}" cannot be lower than 1'.format(low))

    if guesses_allowed < 1:
        die('--guesses "{}" cannot be lower than 1'.format(high))

    if low > high:
        die('--min "{}" is higher than --max "{}"'.format(low, high))

    prompt = 'Guess a number between {} and {} (q to quit): '.format(low, high)
    num_guesses = 0

    while True:
        guess = input(prompt)
        num_guesses += 1

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

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

        num = int(guess)

        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))


# --------------------------------------------------
if __name__ == '__main__':
    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'>
```

Now we need to determine if the user has guessed too many times, if the number if 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)
```

```
# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)

# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    max_len = args.maxlen
    min_len = args.minlen
    max_misses = args.misses
    wordlist = args.wordlist

    if not os.path.isfile(wordlist):
        die('--wordlist "{}" is not a file.'.format(wordlist))

    if min_len < 1:
        die('--minlen must be positive')

    if not 3 <= max_len <= 20:
        die('--maxlen should be between 3 and 20')

    if min_len > max_len:
        die('--minlen ({}) is greater than --maxlen ({})'.format(
            min_len, max_len))

    good_word = re.compile('^[a-z]{' + str(min_len) + ',' + str(max_len) + '}$')
    words = [w for w in open(wordlist).read().split() if good_word.match(w)]
    word = random.choice(words)
    play({'word': word, 'max_misses': max_misses})


# --------------------------------------------------
def play(state):
    """Loop to play the game"""
    word = state.get('word') or ''

    if not word: die('No word!')

    guessed = state.get('guessed') or list('_' * len(word))
    prev_guesses = state.get('prev_guesses') or set()
    num_misses = state.get('num_misses') or 0
    max_misses = state.get('max_misses') or 0

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

    if num_misses >= max_misses:
        bail('You lose, loser!  The word was "{}."'.format(word))

    print('{} (Misses: {})'.format(' '.join(guessed), num_misses))
    new_guess = input('Your guess? ("?" for hint, "!" to quit) ').lower()

    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

    if not re.match('^[a-z]$', new_guess):
        print('"{}" is not a letter'.format(new_guess))
        num_misses += 1
    elif new_guess in prev_guesses:
        print('You already guessed that')
    elif new_guess in word:
        prev_guesses.add(new_guess)
        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
    else:
        num_misses += 1

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


# --------------------------------------------------
if __name__ == '__main__':
    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 decribes 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 them 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 word in open(wordlist).read().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 avialable, 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 is 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 bang. 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 chec 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')]
```

If we `find` "o", it 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:

```
    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, 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.