122COM: Hangman

David Croft

Coventry University

david.croft@coventry.ac.uk

2018

Coventry University

Overview

Aims

Aims

Step by step development of a simple hangman game.

Demonstrating the design strategies that we want to see in you as programmers.

- Functional decomposition.
- Aggregation.
- Encapsulation.
- Information hiding.
- Documentation.
- Testing.

Hangman rules

What are the rules?

- Player 1 picks a word.
- Player 2 quesses individual letters.
- Player 1 tells player 2 if the guesses are correct.
  - and position of correct letters in the word.
- Player 2 has limited number of guesses.
  - 6 guesses if playing the version where you draw the hangman.
- If guesses letter not in word then number of remaining guesses decreases by 1.
- If player 2 correctly quesses the word before the attempts run out then player 2 wins.
- If player 2 fails to correctly quess the word then player 1 wins.

Demo.

Play on whiteboard.

Variables

What is the minimum information we need in a game of hangman?

What is the minimum information we need in a game of hangman?

1. What is the word.
2. What letters have been guessed.
3. How many guesses you get.

All other information can be derived from those 3 pieces.

All other information can be derived from those 3 pieces.

- Which letters guessed correctly.
  - Letters guessed and letters in the word
- Which letters guessed incorrectly.
  - Letters guessed not in the word
- How many guesses remaining.
  - How many guesses you get - incorrect guesses

- Has player 1 won?
  - Guesses remaining == 0
- Has player 2 won?
  - Letters in word - letters guessed == 0
- Is the game over?
  - If player 1 or player 2 has won

So how many variables does our program need and what are they? Want to minimise the number of values we store. What are our variable types?

1. What is the word.

So how many variables does our program need and what are they? Want to minimise the number of values we store. What are our variable types?

1. What is the word.
   - `str`

2. What letters have been guessed

So how many variables does our program need and what are they? Want to minimise the number of values we store. What are our variable types?

1. What is the word.
   - `str`
2. What letters have been guessed
   - Few options here; `str`, `list` or `set`.
3. How many guesses you get

So how many variables does our program need and what are they? Want to minimise the number of values we store. What are our variable types?

1. What is the word.
   - `str`
2. What letters have been guessed
   - Few options here; `str`, `list` or `set`.
3. How many guesses you get
   - `int`

What are we going to call our variables.

- Meaningful names.
- Style conventions.

Variable names

Examples of acceptable variable names. Other options obviously exist.

	Single word	Camelcase	PEP8
Guesses you get	word	guessWord	word_to_guess
Letters guessed	guesses	lettersGuessed	guesses_made
Guesses you get	chances	numOfGuesses	num_of_attempts

UNacceptable variable names.

	Single letters	Random formatting	Inaccurate
Guesses you get	w	guessWord	what
Letters guessed	g	LETTERSGUESSED	wordToGuess
Guesses you get	m	num_Of_Guesses	letters

Style guidelines

Variable need to be consistently named, follow style conventions.

- Tools like `pylint` or `pylint3`.
  - Will show where you code breaks the PEP8 conventions.
  - PEP8 variables are named `with_underscores`.
  - All lowercase, underscores for spaces.
- Don't have to follow PEP8, other conventions exist.
  - I do lots of C++ programming, naming convention is `inCamelCase`.
  - First word all lowercase, spaces removed, following words have first letter capitalised.
- Having a convention and following it more important than which convention.
  - Too difficult to keep switching between PEP8 and C++ conventions.

Minimalism is good

Why minimise the number of variables?

Imagine the code needed to take a turn.

With a minimal number of variables.

```
def guess(letter):
    # sensible approach
    lettersGuessed.add(letter)
```

Minimalism is good

Why minimise the number of variables?

Imagine the code needed to take a turn.

With a minimal number of variables.

```
def guess(letter):
    # sensible approach
    lettersGuessed.add(letter)
```

With a separate variable for everything.

```
def guess(letter):
    # insane approach
    lettersGuessed.add(letter)
    numberofGuesses += 1
    attemptsRemaining -= 1

    if letter in word:
        correctGuesses += 1
    else:
        wrongGuesses -= 1

    if attemptsRemaining == 0:
        player1Won = True
```

- Generally better to store as few values as possible.
  - Derive additional information on the fly.
- Much easier to keep all the information consistent.

Demo.

Step through a game setting variable values.

True, calculating values on the fly can be more computationally expensive that simply caching them.

- Only if you end up using ALL those cached values MULTIPLE times.
- Much harder to debug.
- Much harder to write.
  - Have to remember to update all the values every time.
- Who cares if it takes longer?
  - This isn't performance critical code.

Aggregation

Aggregation

All of these (3) variables are very closely related.

- All for the hangman game.
- All working together.

Should really aggregate them.

Why aggregate?

Why are we aggregate?

- One of the key features of Object Oriented Programming (OOP).
- Imagine we have multiple games going at once.
- Everything stored in one place, together.
- Helps us write generic code.

```
lettersGuessed1 = []
word1 = ""
attempts1 = 6

lettersGuess2 = []
word2 = ""
attempt2 = 6
```

```
class Hangman:
    def __init__(self):
        self.lettersGuessed = []
        self.word = ""
        self.attempts = 6

game1 = Hangman()
game2 = Hangman()
```

Can anyone spot the mistake in the first piece of code?

Can anyone spot the mistake in the first piece of code?

- The 2nd `game_won()` call is using `attempt1` not `attempt2`.
- Aggregation means fewer bugs.

```
def game_won(word, lettersGuessed, attempts):
  if set(word) in lettersGuessed:
    return "Player 2"
  if len(lettersGuessed) >= attempts:
    return "Player 1"

  return None

game_won( word1, lettersGuessed1, attempt1 )
game_won( word2, lettersGuessed2, attempt1 )
```

```
def game_won(game):
  if set(game.word) in game.lettersGuessed:
    return "Player 2"
  if len(game.lettersGuessed) >= game.attempts:
    return "Player 1"

  return None

game_won( game1 )
game_won( game2 )
```

Functional decomposition

Functional decomposition

On the previous slides we had the game won code in a separate function.

Breaking up the separate parts of a program is called functional decomposition. A **VITAL** programming skill.

- Reusable code.
  - Less code.
  - = fewer bugs.
- Maintainable code.
- Testable code.
  - = fewer bugs
- Collaborative code.

What actions can be performed in a game of hangman?

- Guess a letter.
  - Needs to know the letter.
- New game.
  - Needs to know the new word.

What information can you get from a game of hangman?

- Is the game over?
- Is the game won?
- What letters have you already guessed?
- How long is the word?
- Guesses remaining.

Each of these actions can be written as a separate function.

- Each is separate so each can be tested separately.
  - Easy testing.
- Each is separate so each can be written separately.
  - Easy collaboration.
- Reuse.
  - If need to do same thing multiple times.
- Single responsibility principle.
  - Program logic is much clearer.

```
game = Hangman()

while len([ i for i in game.lettersGuessed if i not in game.word ]) >= game.attempts:
  print( "".join([ "_" if i not in game.lettersGuessed else i for i in game.word ]) )

  letter = input('Enter guess: ')

  if letter not in game.lettersGuessed:
    game.lettersGuessed.append( letter )

print( "".join([ "_" if i not in game.lettersGuessed else i for i in game.word ]) )

winner = 2 if [i for i in self.word not in game.lettersGuessed] == [] else 1
print( "Player {} won!".format( winner ) )
```

Before.

```
game = Hangman()

while not game_over( game ):
    display( game )

    letter = input('Enter guess: ')
    guess( game, letter )

display( game )

winner = who_won( game )
print( "Player {} won!".format( winner ) )
```

After.

Coventry University

Encapsulation

Functions

Have aggregated the variables.
Have a series of functions to interact with the variables.
Functions only deal with these game variables. Why are the variables and the functions that deal with them separate?

- Encapsulate the functions.
  - Member functions.
  - Neater code, the variable and the functions that deal with them together.
- Variables in classes are attributes of that class.
  - Private attributes cannot be accessed by code outside the class.
  - Functions outside the class can't read/write them.
  - Functions inside the class can read/write them.
- Private attributes are protected.
  - Can only interact with them in approved ways.

Information hiding

```
class Hangman:
  def __init__( self ):
    self.lettersGuessed = []

def guess( game, letter ):
  game.lettersGuessed.append(letter)

game1 = Hangman()
guess( game1, 'Z' )
```

Before.

```
class Hangman:
  def __init__( self ):
    self.__lettersGuessed = []

  def guess( self, letter ):
    self.__lettersGuessed.append( letter )

game1 = Hangman()
game1.guess( 'Z' )
```

After.

Information hiding

Force attribute interactions through approved functions.

- Can be sure everything is done the correct way.

```
game1 = Hangman()

# works
guess( game1, 'Z' )

# but so does this
game1.lettersGuessed.append( "a fish" )
```

Before.

```
game1 = Hangman()

# only way to make a guess
game1.guess( 'Z' )

# won't work
game1.lettersGuessed.append( "a fish" )
game1.__lettersGuessed.append( 42 )
```

After.

Reuse

Encapsulating everything and forcing all interactions through member functions means easily reused class.

Get user input code changes but game code remains the same.

- Command line version

```
letter = input("Enter your guess")
game1.guess(letter)
```

- GUI version

```
for letter in string.ascii_uppercase:
    button = tk.Button(root, text=letter, \
                       lambda: game1.guess(letter))
    button.pack()
```

- Network server version

```
msg = connection.recv(1024)
header, value = msg.split(" ")
if header == "guess":
    game1.guess(value)
```

Coventry University

Testing

Robustness

Code works now but only just.

- Invalid inputs.
  - Parameter and value validation.
- Exception raising.
- Exception handling.

Break it!

What inputs could we give to this function that would cause problems?

```
lettersGuessed = []

def guess(letter):
  lettersGuessed.append(letter)
```

Break it!

What inputs could we give to this function that would cause problems?

```
lettersGuessed = []

def guess(letter):
  lettersGuessed.append(letter)
```

- More than one letter.
  - `"abc"`
- Not a letter.
  - `42`, `[42,69]`
- Combinations of upper and lowercase letters.
  - Are `'A'` and `'a'` different guesses?
- Letters that have already been guessed.
  - `'A'`, `'A'` and `'A'`.

Demo.

Give your code to another person and have them try and break it.

Quality Assurance (QA) testing.

Fix it!

```
lettersGuessed = []

def guess(letter):
  lettersGuessed.append(letter)
```

Before.

```
lettersGuessed = set()

def guess(letter):
  if not isinstance(letter, str):
    raise TypeError("Not a string")
  elif len(letter) != 1:
    raise ValueError("Not a single letter")
  elif letter not in string.ascii_letters:
    raise ValueError("Not a letter")

  letter = letter.upper()
  lettersGuessed.add(letter)
```

After.

Could run all the tests manually.

- Verification tests.
  - Have I fixed the problem?
- Regression tests.
  - Have I caused any new problems?
  - On average every 3 bugs fixed == 1 new bug.

Manually running tests takes ages.

- Automated testing.
  - Write code to test your code.

Could store them in separate file. Could store them in same file.

```python
if __name__ == "__main__":
    # tests go here
```

```
if __name__ == "__main__":
    # test new game
    game = Hangman()
    assert game.chances() == 7, /
        "New game but wrong number of guesses remaining"

    # test guess code
    game = Hangman()    # fresh instance for each test

    before = game.remaining_chances()
    game.guess("A")
    after = game.remaining_chances()

    assert after == before -1, /
        "Guessed a letter but remaining_chances did not decrease"
```

The End