Writing a guessing game

COR 100.13 / Year 1 - Game programming with Python

Marcus Birkenkrahe

October 27, 2024

Defining the game

- We're going to bring the last few topics together in a complete little game script, a Guess the Number game.
- In this game, the computer will think of a secret number from 1 to 20 and ask the user to guess it. After each guess, the computer will tell the user whether the number is too high or too low. The user wins if they can guess the number within six tries.
- The game uses many new Python tools:
  1. random numbers
  2. repeating code chunks
  3. grouping and indenting code
  4. selecting choices based on conditions
  5. converting values to different data types
  6. breaking out of loops

Planning the game

- This example also demonstrates an exemplary solution path:
  1. Understand what's asked from you (**requirements**)
  2. Understand what the program needs from you (**input**)
  3. Understand what's the result supposed to look like (**output**)
  4. Plan the process without syntax (**pseudocode**)

5. Create a process **diagram** (with commands)
6. Code the Python program (**source code**)
7. Run, test and debug the source code (**production code**)
8. Fix pseudocode/diagram accordingly (**feedback**)
9. Identify **extensions** (other things you might like)
10. Implement extensions (repeat steps 4-8).

- When you run the program, the output should look like this (tinyurl.com/guess-the-number-output):

```
Enter number between 1 and 20:
Take a guess: 10
Your guess is too high.
Take a guess: 2
Your guess is too low.
Take a guess: 8
Your guess is too high.
Take a guess: 3
Your guess is too low.
Take a guess: 7
Good job! You guessed my number in 5 guesses!
```

Figure 1: Desired output of guessTheNumber.py

- The program should generate a random number between 1 and 20.
- Enter the source code into the IDLE file editor, or into Colab, and save as `guessTheNumber.py`.
- Solution path/pseudocode (code highlighted)

1. `import random` module.
2. Generate a (secret) `random` number.
3. Store number in variable `num`.
4. Set `attempt` counter (number of guesses) to `0`.

5. Get `input` number `guess` from user.
6. Increase `attempt` by 1
7. Check if `guess` is the same as `num`
8. `print` success message and `attempt` value
9. End program
10. Otherwise, check if `guess` is smaller than `num`
11. `print` information
12. Otherwise, check if `guess` is larger than `num`
13. `print` information
14. Return to step 3

- The BPMN Process diagram is fairly complicated compared to the previous example (tinyurl.com/guess-the-number-png):

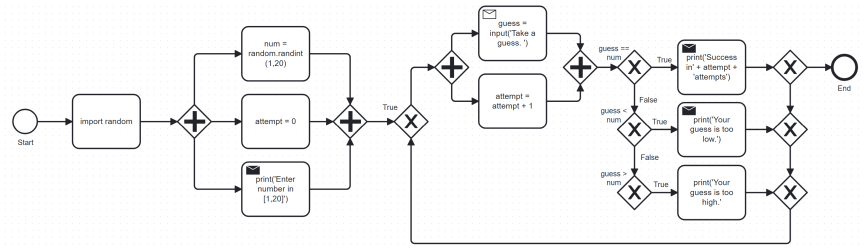

Figure 2: Flow diagram for guessTheNumber.py

- Solution Python code (16 + 5 lines):

```
# import random module
import random
# pick random number between 1 and 20
num = random.randint(1,20)
# set attempts counter to 0
attempt = 0
# ask user for number guess
print('Enter number between 1 and 20: ')
# infinite loop until number is guessed
while True:
    guess = int(input('Take a guess: '))
    attempt = attempt + 1
```

```
if guess < num:
    print('Your guess is too low.')
    continue
elif guess > num:
    print('Your guess is too high.')
    continue
else:
    print('Good job! You guessed my number in ' + str(attempt) + ' guesses!')
    break
```

Generating random numbers

- To generate a secret guess, we use a (pseudo-) random number generator. In Python, such a generator is contained in the `random` package.
- To make this package available, you need to *install* it to your computer and then *load* it in the Python session where you need it.
- In Colab, the package is already installed and only has to be loaded:

  ```
  import random
  ```

- We can now pick a random number between 1 and 20 using the `randint` function in `random`: run the following code multiple times to see how it works.

  ```
  print(random.randint(1,20))
  ```

- The `.` operator accesses the `random` package. You remember that we loaded a graphics package, `matplotlib.pyplot` earlier, and gave it an alias, `plt`:

  ```
  import matplotlib.pyplot as plt
  ```

- To access the `plot` function in the package, we called the function twice: first to pass four points for plotting, and then to make the plot appear on the screen:

  ```
  plt.plot([1,2,3,4])
  plt.show()
  ```

- You need randomness in many games - even board games use dice, and many game actions, e.g. by NPCs, are randomized.

Repeating code

- The next part of the code that may be new to you if you never programmed before is the line `while True:`
- This is an infinite loop: the `while` command enters the loop followed by a test. The generic form of the command is:

```
while [test]:
    # do something
```

- The result of the test is either `True` in which case the loop is entered, or `False`, in which case it is left again without doing anything.
- Let's look at a few examples:

```
i = 1
while i < 3:
    i = i + 1
    print(i)
```

```
2
3
```

- Let's analyze:
  1. Here, `i` is set to 1. When the `while` is encountered, `i < 5` is tested. Since it's `True`, the statements in the loop body are run: `i` is increased to 2, and printed.
  2. The loop is entered a second time: the test `2 < 5` is still `True`, `i` is increased to 3, and printed.
  3. The loop is entered a third time: the test `3 < 5` fails - it is evaluates to `False`, and the loop commands are not executed.
- **As a challenge**, change the `while` loop so that it starts at `i = 5` and tests if `i > 0`, so that the output is: `4 3 2 1 0`.

```
i = 5
while i > 0:
    i = i - 1
    print(i)
```

```
4
3
2
1
0
```

- Coming back to our game: If the test reads `True` then the condition *never* fails and the loop will keep running forever!
- To stop the game inside an infinite loop, we must take extra measures: we must `break` out of the loop.
- Here is an example: This loop runs exactly once and then exits because of the `break` command.

```
while True:
    print("Infinite loop!")
    break
print("Done!")
```

- The next one runs until `q` is entered. It prints the message to the screen and then halts waiting for input:

```
while True:
    print("Infinite loop...until you type q")
    if input()=='q': break
    print("Done!")
```

- The last example checks a condition after the `if` keyword: this is called a *conditional statement*. Seen through process model eyes, this last code chunk looks like this (tinyurl.com/conditional-statement):

  [width=.9]../img/conditional$_s$*tatement*

Checking conditions

- The core of the infinite game loop also has a conditional statement. Instead of one check, it has two: namely, if the user's guess, stored in `guess`, is greater or smaller than the computer's (secret) number:

```
if guess < num:
    print('Your guess is too low.')
    continue
elif guess > num:
    print('Your guess is too high.')
    continue
else:
    print('Good job! You guessed my number in ' + str(attempt) + ' guesses!')
    break
```

This is what happens inside the loop:

1. If the guess is smaller than the computer's number, the user is told that it is, and we `continue` with another guess.
2. If the guess is greater than the computer's number, the user is told that it is, and we `continue` with another guess.
3. If the guess is neither smaller nor greater than the computer's number, we must have guessed it: then we print the result and `break` out of the loop to finish.

- Challenge:
    1. Define an integer `foo` with a value in [0,100]
    2. Write a selection statement that tests if `foo` is
        - smaller than 50 (`num < 50`)
        - greater than 50 (`num > 50`)
    3. Print a message based on the range the number falls into that includes the number.
- Solution:

```
foo = 60
if (foo <50):
    print(str(foo) + " is smaller than 50")
elif (foo > 50):
    print(str(foo) + " is greater than 50")
```

```
60 is greater than 50
```

- Modifications:

  1. How would you test if the number is 50 exactly?
  2. How would you test if the number is either smaller than 10, or between 10 and 50, or greater than 50?
  3. How would you test if the number is negative?

- Solution: If the number is 50 exactly.
- Solution: If the number is smaller 10, in the range 10 to 50, or greater than 50.
- Solution: If the number is negative .

Getting the user's number

- At the start of the loop, we get the user's guess and store it in the variable `guess`.
- We get this number from the keyboard with `input`:

  ```
  guess = int(input())
  ```

- You notice that we did not write `guess = input()`. Why? Let's see:

  ```
  print('Enter a number between 1 and 20:')
  guess = input()
  print(guess < 20)
  ```

- When you run this code, you get a `TypeError`:

  ```
  TypeError: '<' not supported between instances of 'str' and 'int'
  ```

- The problem is that you cannot compare a string (`str`) and an integer (`int`) - and with `input` you can only import strings from the keyboard.
- To fix this, you must *convert* the string to an integer. This only works, of course, if the converted string can be recognized as a number: it works for `"2"` but not for `"a"` or `"2 + 2"`:

  ```
  print(int("2"))
  print(int("2 + 2"))
  print(int("a"))
  ```

```
2
```

- So `guess` holds not the string value of the user's number but the integer value, which can be compared with the computer's number.

Printing the result

- Fortunately, we have already understood the concept of conversion: in the printout of the result, another conversion takes place, but this time the other way around, from integer to string:

```
print('Good job! You guessed my number in ' + str(attempt) + ' guesses!')
```

- In the case of `str`, any number can be turned into a string:

```
print(str(1e+3))
print(str(0.001))
```

```
1000.0
0.001
```

- One way of testing if a value is a string or a number is by concatenating it with another string:

```
print("The number is " + str(0.001))
print("The number is " + 0.001)
```

```
The number is 0.001
```

- As before, the second command fails with a `TypeError`:

```
TypeError: can only concatenate str (not "float") to str
```

Putting it all together

- In the next code block, let's assemble the whole program, add a condition at the end of the loop to offer quitting with 'q', and print the winning number of attempts using the `str` cast function:

```
import random  # import random module
num = random.randint(1,20) # pick random number in (1,20) - computer's number
attempt = 0 # initialize number of attempts
print("Guess my number! Enter a number between 1 and 20!")
while True:  # start the infinite loop
    guess = int(input('Take a guess: ')) # Prompt user input & store in guess
    attempt = attempt + 1 # increase number of attempts
    if guess < num: print("Too low")
    elif guess > num: print("Too high")
    else:
        print("Attempts: " + str(attempt))
        break
    if input("Enter 'q' to quit (or RET to continue: ")=='q': break
```

- You find yet another solution in Sweigart's book (chapter 3):

```
# This is a Guess the Number game.
import random
guessesTaken = 0
print('Hello! What is your name?')
myName = input()
number = random.randint(1, 20)
print('Well, ' + myName + ', I am thinking of a number between 1 and 20.')
for guessesTaken in range(6):
    print('Take a guess.') # Four spaces in front of "print"
    guess = input()
    guess = int(guess)
    if guess < number:
        print('Your guess is too low.') # Eight spaces in front of "print"
    if guess > number:
        print('Your guess is too high.')
    if guess == number:
       break
if guess == number:
```

```
        guessesTaken = str(guessesTaken + 1)
        print('Good job, ' + myName + '! You guessed my number in ' +
              guessesTaken + ' guesses!')
    if guess != number:
        number = str(number)
        print('Nope. The number I was thinking of was ' + number + '.')
```

Program extensions and lessons learnt

- Program extensions:
  1. Make program safe against no/wrong input (exception handling): currently, it terminates with an error if a floating-point number or a letter or nothing is entered by mistake.
  2. Exchange the infinite `while` loop by a `for` loop with a set number of allowed guesses (most games don't go on forever).
- What's important to remember:
  1. For best productivity and learning, follow a solution path - don't just "code away"
  2. For best learning effects find different solutions to the same problem.
  3. For best results, handle exceptions. Balance exception handling with usability and performance.
  4. There is always more than one solution, usually many. There is no best solution to a programming problem that satisfies all requirements, even the unspoken ones, equally well.

Summary

- Expressions as part of an `if` or `while` statement are conditions. They evaluate to Boolean (truth) values.
- `break` and `continue` are flow control statements to break out of a loop or go back to the start of the loop.
- `print` and `input` serve the standard output (stdout) and the standard input (stdin) data stream, or output (e.g. to the screen) and input (e.g. from the keyboard).

- `int` and `str` are functions that convert strings and numbers into integers and strings, respectively.