Print to PDF

Week 4: Iteration and string

Contents

Important

You must start by downloading the script

> download_week04.py

Right-click and select "Save link as...". You may need to confirm the download in your browser.

Run this script in VSCode to download the code snippets and tests you need for this week and the project. The script does not need to be located in a specific place on your computer.

If you get an error from the script, please follow the instructions in the error message. If you give up, proceed to the exercises (which you can still do) and ask a TA once they arrive.

The script should end with `Successfully completed!`, which means you now have this week's tests.

Videos (Watch this before the other videos)

What I am going to cover is:

- A bit about the course
- A few additional installation problems I saw last week (test discovery)
- Exercise walkthrough and project hints

I have tried to make this week debugger-intensive, meaning I use the debugger in all exercises – what the debugger does is obviously be covered in the videos, but basically if you learn to use two more buttons in VS Code, you will become much more productive!

Please note the videos are not a stand-in for the lectures or Think Python, and you should prioritize working on the exercises yourself. :-).

Skip to main content

Video 4a: Watch this first

Video 4b: A few more common problems (seeing tests)

Note

In most of the files this week you will see this at the end:

```
if __name__ == "__main__":
    some_function()
```

Whatever you write inside the if statement will be executed when you run the file directly, but not when you import it. This is useful if you want to try out your code directly and make sure any testing you do won't interfere with the provided tests.

Exercise 4.1: Square roots

Video 4.1: Exercise walkthrough

In this exercise, you will compute the square root of a number a using a simple algorithm: Newton's method. The method is described in [Dow16], section 7.5. The method is iterative, meaning that it repeatedly applies the same operation to get closer and closer to the answer. The method is as follows:

- Start with a guess x
- Compute a new guess y using the formula:

$$y = \frac{x + \frac{a}{x}}{2}$$

- Repeat the previous step until $x = y$ (or the difference between x and y is smaller than a given tolerance ϵ).

Try your function with different values to see if it computes the square root correctly

Skip to main content

The tests for the function `square_root` can be run by inserting your solution into the file `cp/ex04/mathematics.py`.

cp.ex04.mathematics.square_root(*a*)

Compute the square root, see section 7.5 in Think Python.

Parameters:

a (`float`) – A number to compute the square root of.

Return type:

`float`

Returns:

$\sqrt{a}$.

Exercise 4.2: Efficiently computing pi

Video 4.2: Exercise walkthrough

Now you will estimate the value of π using Ramanujan's approximation:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Further details can be found in [Dow16], exercise 7.3. Write some code that estimates the value of π using this formula. You can use a loop to compute the sum, and you should stop when the last term is smaller than a given tolerance ϵ. You can use the factorial function from the math module to compute the factorials.

```
>>> import math
>>> math.factorial(4) # Factorials, i.e. 4! = 4*3*2
24
```

Check if your solution is actually π.

The tests for the function `ramanujan` can be run by inserting your solution into the file `cp/ex04/mathematics.py`.

cp.ex04.mathematics.ramanujan()

Compute the Ramanujan approximation of π using a sufficient number of terms.

Return type:

`float`

Returns:

A high-quality approximation of π as a `float`.

Exercise 4.3: Is it a palindrome?

Video 4.3: Exercise walkthrough

A **palindrome** is a word that reads the same backwards as forwards, such as *madam* or *racecar*. Your job is to check whether a given word is a palindrome or not. Your function should return `True` if the word is a palindrome and `False` otherwise. Example:

Skip to main content

```
>>> is_palindrome('madam')
True
>>> is_palindrome('gentleman')
False
```

You can test the function by inserting your solution into the file `cp/ex04/palindrome.py`.

cp.ex04.palindrome.is_palindrome(*word*)

Check if `word` is a palindrome.

Parameters:

word (`str`) – The word to check

Return type:

`bool`

Returns:

`True` if input is a palindrome and otherwise `False`

Exercise 4.4: The last bug

Video 4.4, 4.5: Exercise walkthrough

Now you will write some nice messages in the console. This can be nice to add to your code. First, you will write a message that you have written the last bug in your code. The function should print out a nice sign surrounded by pipes and dashes that looks like this:

```
>>> last_bug()
-----------------------------
| I have written my last bug |
-----------------------------
```

You can test the function by inserting your solution into the file `cp/ex04/bug.py`.

cp.ex04.bug.last_bug()

Write a nice message enclosed by lines and pipes that clearly indicate you have written your last bug.

The function should print out the following three lines in the console:

```
-----------------------------
| I have written my last bug |
-----------------------------
```

Exercise 4.5: Writing pretty signs to the console

Now that you are done with the bugs, let's write a function to print out other pretty enclosed signs to the console. Write some code that encloses any (one-line) message in dashes (-) and pipes (|). For instance:

```
>>> nice_sign("Hello World!")
---------------
| Hello World! |
---------------
```

Skip to main content

`cp.ex04.bug.nice_sign(msg)`

Print the input string as a nice sign by enclosing it with pipes.

Note that the input message can vary in length.

```
---------------
| {input msg} |
---------------
```

Parameters:

msg (`str`) – The message to enclose.

You can test the function by inserting your solution into the file `cp/ex04/bug.py`.

Exercise 4.6: Parenthesis matching

Video 4.6: Exercise walkthrough

Suppose you were writing a program that had to validate mathematical expressions such as `"3x(y+x)"`. One task this program would need to accomplish was to test if the parentheses matched in the expression. Write a function which accepts a mathematical expression as a `str` and returns `True` if the opening and closing parentheses match. For instance:

```
>>> matching('3x(y+x)')
True
>>> matching('3x(y+(x-1))')
True
>>> matching('3xy+x)')
False
```

You can test the function by inserting your solution into the file `cp/ex04/parenthesis.py`.

`cp.ex04.parenthesis.matching(expression)`

Tell if the parenthesis match in a mathematical expression.

For instance, the parenthesis match in `"3x(y-1)(3y+(x-1))"` but not in `"3x(y-4))"`

Parameters:

expression (`str`) – An expression containing zero or more parenthesis.

Return type:

`bool`

Returns:

`True` if the number of open/close parenthesis match, otherwise `False`

Exercise 4.7: The inner-most expression (Challenge problem!)

Video 4.7: Exercise walkthrough

Continuing the previous example, suppose we wanted to evaluate a complicated mathematical expression specified as a `str` which contains parentheses such as `"3(x+(y-1))"`. One approach would be to first find the *inner-most substring*, i.e. a substring of the expression enclosed in parentheses but not containing parentheses. In this case, one such example is `"y-`

Skip to main content

```
>>> find_innermost_part('(y+x)')
'y+x'
>>> find_innermost_part('3x(y+(x-1))')
'x-1'
>>> find_innermost_part('3x((y+2)y+x)')
'y+2'
```

Note that there may be multiple inner-most substrings, in which case you should just return one of them.

You can test the function by inserting your solution into the file `cp/ex04/parenthesis.py`.

cp.ex04.parenthesis.find_innermost_part(*s*)

Find the innermost part of a mathematical expression.

The innermost part is a substring enclosed in parentheses but not containing parentheses. For instance, given `"3(x+(4-y^2))"`, then `"4-y^2"` is an inner-most part. The parenthesis can be assumed to match.

Parameters:

s (`str`) – The mathematical expression as a `str`

Return type:

`str`

Returns:

The innermost part as a `str`

Exercise 4.8: Matching parenthesis

Video 4.8: Exercise walkthrough

Given a `str` containing parentheses, for instance `"(()())"`, this function should return an index `i`, such that when the string is split at `i`, the number of opening parentheses `(` in the left-hand part equals the number of closing parentheses `)` in the right-hand part. For instance, if `i=3`, the expression is split into the right and left hand parts:

```
>>> s = "(()())"
>>> i = 3
>>> s[0:i]
'(()'
>>> s[i:]
'())'
```

In this case, the left-hand part contains `2` opening parentheses and the right-hand side contains `2` closing parentheses so `i` is the right index. Here are a couple of additional examples:

```
>>> find_index_of_equality("(()())")
3
>>> find_index_of_equality('()')
1
>>> find_index_of_equality('(()')
1
>>> find_index_of_equality('()))')
3
```

You can test the function by inserting your solution into the file `cp/ex04/parenthesis.py`.

cp.ex04.parenthesis.find_index_of_equality(*expression*)

Skip to main content

Given an expression containing opening and closing parenthesis, for instance `"(()())"`, this function should return an index `i`, such that when the string is split at `i`, the number of opening parenthesis `(` in the left-hand part equal the number of closing parenthesis `)` in the right-hand part. For instance, if `i=2`, the expression is split into the right, left hand parts:

- `"(()"`
- `"())"`

In this case the left-hand part contains `2` opening parenthesis and the right-hand part `2` closing parenthesis so `i` is the right index. Similarly, for `"()"`, the answer would be `1`.

Parameters:

expression (`str`) – An expression only consisting of opening and closing parenthesis.

Return type:

`int`

Returns:

The index `i` as an int.

Exercise 4.9: Getting the dialogue

Video 4.9: Exercise walkthrough

Imagine you are training an AI on movie dialogue from movie scripts represented using strings. The dialogue can be recognized by being enclosed by double single-quotation marks, i.e. if the function is called with:

- `"''My dear Mr. Bennet,'' said his lady to him one day, ''have you heard that Netherfield Park is let at last?''"`

then the two dialogue strings are `"My dear Mr. Bennet,"` and `"have you heard that Netherfield Park is let at last?"`. Your job is to write a function which accepts a manuscript represented as a `str`, and print out **all** the lines of dialogue on separate lines. For instance:

```
>>> dialogue = "''My dear Mr. Bennet,'' said his lady to him one day, ''have you heard that Netherfield Park is
>>> print_the_dialogue(dialogue)
My dear Mr. Bennet,
have you heard that Netherfield Park is let at last?
```

You can test the function by inserting your solution into the file `cp/ex04/parenthesis.py`.

`cp.ex04.parenthesis.print_the_dialogue(s)`

Print all dialogue in a manuscript.

Given a manuscript (as a `str`), this function will find all lines of dialogue to the console, one line of dialogue per printed line. Dialogue is enclosed by double ticks, i.e. this `str` contains two pieces of dialogue: `"''My dear Mr. Bennet,'' said his lady to him one day, ''have you heard that Netherfield Park is let at last?''"`

Parameters:

s (`str`) – The manuscript as a `str`.

Skip to main content

Exercise 4.10: Common prefixes

Video 4.10: Exercise walkthrough

In this exercise, you will implement a function that finds the longest common prefix of two words. For instance, the longest common prefix of `"coffee"` and `"covfefe"` is `"co"`.

```
>>> common_prefix("coffee", "covfefe")
'co'
>>> common_prefix("hat", "cat")
''
```

cp.ex04.prefix.common_prefix(*word1*, *word2*)

Return the longest string so that both `word1`, and `word2` begin with that string.

Parameters:

- **word1** (`str`) – First word
- **word2** (`str`) – Second word

Return type:

`str`

Returns:

The longest common prefix.

You can test the function by inserting your solution into the file `cp/ex04/prefix.py`.

Exercise 4.11: Common prefixes of three words

Video 4.11: Exercise walkthrough

Now you should extend your function from before to take three words and return the longest common prefix of all three words. For instance:

```
>>> common_prefix3("coffee", "covfefe", "cow")
'co'
>>> common_prefix3("cat", "covfefe", "cow")
'c'
```

cp.ex04.prefix.common_prefix3(*word1*, *word2*, *word3*)

Return the longest string so that both `word1`, `word2`, and `word3` begin with that string.

Parameters:

- **word1** (`str`) – First word
- **word2** (`str`) – Second word
- **word3** (`str`) – Third word

Return type:

`str`

Returns:

The longest common prefix.

Skip to main content

Project exercises: Hangman! Introduction

Project 2b: Getting started

In the next exercises, you will implement an interactive variant of the game **hangman**. If you are unfamiliar with the rules, you can find them on Wikipedia. Our approach to implementing the game is to break the problem into several smaller functions, so even if you are a bit fuzzy on the complete game, that should not prevent you from getting started. All completed functions should be put in the file `cp/ex04/hangman.py` to run the supplied tests.

The function we eventually want to implement is called `hangman()`. It is given two input arguments (a word to guess, as a `str`, and a number of guesses available, as an `int`) and then it allows you to sequentially guess at letters until you either win or lose. In other words, this is how you can start the game:

Starting Hangman. The secret word is "cow"

```
# lecture4/hangman_examples.py
hangman("cow", guesses=6)
```

After the game is launched, the user will input letters using the keyboard to guess the word. This shows a complete interaction where the user inputs `o`, `x`, `w`, `c`, and therefore ends up winning the game:

Example hangman console interaction

```
Hangman! To save Bob, you must guess a 3 letter word within 6 attempts.
----
You have 6 guesses left.
The available letters are: abcdefghijklmnopqrstuvwxyz. Guess a letter and press enter: o
Good guess: _ o_
----
You have 5 guesses left.
The available letters are: abcdefghijklmnpqrstuvwxyz. Guess a letter and press enter: x
Oh no: _ o_
----
You have 4 guesses left.
The available letters are: abcdefghijklmnpqrstuvwyz. Guess a letter and press enter: w
```

Skip to main content

```
You have 3 guesses left.
The available letters are: abcdefghijklmnpqrstuvyz. Guess a letter and press enter: c
Success! You guessed 'cow' in 4 tries.
Your score is 9
```

Your implementation should eventually be able to reproduce this interaction character by character.

Exercise 4.12: Hangman! Is the word guessed?

If you think about it, a hangman program must keep track of

- What the true word is, `"cow"`,
- What letters have been guessed currently.

This is called the data structure of the program, and will be the most important choice you make as a programmer. In our game, it is relatively straightforward:

- We store the true word, `"cow"`, as a `str`
- We store the letters that have been guessed as another `str`: `"ewoc"`

Thus, having made that choice, we can break the problem into smaller functions that we put together at the end to implement `hangman()`.

The first function you should implement is `is_word_guessed()`. This function is given the word to guess, `secret_word`, and the letters guessed and should return `True` if the word has been guessed and otherwise `False`. An example:

```
>>> is_word_guessed("cow", "owqc") # Yes, all letters are guessed
True
>>> is_word_guessed("cow", "owq") # No, we did not guess on c yet
False
```

cp.ex04.hangman.is_word_guessed(*secret_word*, *letters_guessed*)

Determine if the word has been guessed.

Parameters:

- **secret_word** (`str`) – The word to guess
- **letters_guessed** (`str`) – A `str` containing the letters that have currently been guessed

Return type:

`bool`

Returns:

True if and only if all letters in `secret_word` have been guessed.

Exercise 4.13: Hangman! What letters are available?

A common problem in hangman is that it can be hard to tell which letters have been guessed and which have not been guessed. To fix that, we build a function which accepts a string of guessed letters and returns another string of letters that are available. This shows a basic interaction with the method:

```
>>> get_available_letters("bcdifhklmnopqrsyz") # We only have a few letters available
'aegjtuvwx'
>>> get_available_letters("") # This will show all letters in the game
'abcdefghijklmnopqrstuvwxyz'
```

Skip to main content

`cp.ex04.hangman.get_available_letters(letters_guessed)`

Return the letters which are available, i.e. have not been guessed so far.

The input string represents the letters the user have guessed, and the output should then be the lower-case letters which are not contained in that string. The function is used to show the user which letters are available in each round.

Parameters:

letters_guessed (`str`) – A *str* representing the letters the user has already guessed

Return type:

`str`

Returns:

A *str* containing the letters the user has not guessed at yet.

Exercise 4.14: Hangman! Format the current guess

Our next task is to tell the user which letters in the word have been guessed and which have not been guessed. This is done in the function `get_guessed_word()`. As before, it accepts a secret word and the letters guessed. Then it returns a string, corresponding to the secret word, but where the letters in the secret word that have **not** been guessed are replaced with `"_ "` (i.e., underscore followed by space). This looks as follows:

```
>>> get_guessed_word("cow", "c")
'c_ _ '
>>> get_guessed_word("species", "cei")
'_ _ ecie_ '
```

`cp.ex04.hangman.get_guessed_word(secret_word, letters_guessed)`

Format the secret word for the user by removing letters that have not been guessed yet.

Given a list of the available letters, the function will replace the letters in the secret word with '_ ' (i.e., a lower-case followed by a space). For instance, if the secret word is `"cat"`, and the available letters are `"ct"`, then the function should return `"c_ t"`.

Parameters:

- **secret_word** (`str`) – A `str`, the word the user is guessing
- **letters_guessed** (`str`) – A `str` containing which letters have been guessed so far

Return type:

`str`

Returns:

A `str`, comprised of letters, underscores (_), and spaces that represents which letters in secret_word have been guessed so far.

Exercise 4.15: Hangman! Putting it all together

You are now ready to implement a basic version of the hangman game. The hangman game is a function that accepts the secret word and a number of guesses, and then lets the user guess, one character at a time, what the secret word is.

To solve the exercise, you should implement the function `hangman()`. The function is launched as follows:

```
Starting Hangman with the word to guess and a number of guesses
```

Skip to main content

```
# lecture4/hangman_examples.py
hangman("cow", guesses=6)
```

The first thing the game should `print` is a welcome message:

What the program displays right after launch

```
Hangman! To save Bob, you must guess a 3 letter word within 6 attempts.
----
You have 6 guesses left.
The available letters are: abcdefghijklmnopqrstuvwxyz. Guess a letter and press enter:
```

Inputs

On launch, the game waits for the user to input a character (followed by enter). In this example, if we input `w` followed by enter we get this output:

Inputting a character

```
Hangman! To save Bob, you must guess a 3 letter word within 4 attempts.
----
You have 4 guesses left.
The available letters are: abcdefghijklmnopqrstuvwxyz. Guess a letter and press enter: w
Good guess: _ _ w
----
You have 3 guesses left.
The available letters are: abcdefghijklmnopqrstuvxyz. Guess a letter and press enter:
```

If the character the user inputs is not in the secret word, for instance if we input `q`, the program should indicate this as follows:

Inputting a character not in the word

```
Hangman! To save Bob, you must guess a 3 letter word within 4 attempts.
----
You have 4 guesses left.
The available letters are: abcdefghijklmnopqrstuvwxyz. Guess a letter and press enter: q
Oh no: _ _ _
----
You have 3 guesses left.
The available letters are: abcdefghijklmnoprstuvwxyz. Guess a letter and press enter:
```

Winning

If we type in all the letters of the word within the number of guesses, we win. The game should in that case display the number of points, computed as:

$$\{\text{Points}\} = \{\text{Letters in the secret word}\} \times \{\text{Number of remaining guesses}\}$$

The following example illustrates this interaction in the case where no mistakes are made:

Example of a won game

```
Hangman! To save Bob, you must guess a 3 letter word within 4 attempts.
----
You have 4 guesses left
```

Skip to main content

```
----
You have 3 guesses left.
The available letters are: abcdefghijklmnpqrstuvwxyz. Guess a letter and press enter: w
Good guess: _ ow
----
You have 2 guesses left.
The available letters are: abcdefghijklmnpqrstuvwxyz. Guess a letter and press enter: c
Success! You guessed 'cow' in 3 tries.
Your score is 6
```

Losing

When all guesses are used up, the game is lost, and the following message should be displayed:

Example of a lost game

```
Hangman! To save Bob, you must guess a 3 letter word within 4 attempts.
----
You have 4 guesses left.
The available letters are: abcdefghijklmnopqrstuvwxyz. Guess a letter and press enter: c
Good guess: c_ _
----
You have 3 guesses left.
The available letters are: abdefghijklmnopqrstuvwxyz. Guess a letter and press enter: o
Good guess: co_
----
You have 2 guesses left.
The available letters are: abdefghijklmnpqrstuvwxyz. Guess a letter and press enter: m
Oh no: co_
----
You have 1 guesses left.
The available letters are: abdefghijklnpqrstuvwxyz. Guess a letter and press enter: q
Oh no: co_
Game over :-(. Your score is 0 points.
```

Error handling

Whenever you ask a user for an input, you should be prepared for getting an input back which is poorly formatted or does not match the expectations of your program. In our case, we should expect that:

- The user input zero or more than one letter
- The user input a letter which has already been tried
- The user input a symbol which is not on the list of available letters, for instance `#`

We will deal with all these situations in the simplest way possible: When the user's input does not match our expectation, the game immediately terminates, and the user loses. For instance, suppose the user inputs `c` twice:

The program quits whenever the user makes an error

```
Hangman! To save Bob, you must guess a 3 letter word within 4 attempts.
----
You have 4 guesses left.
The available letters are: abcdefghijklmnopqrstuvwxyz. Guess a letter and press enter: c
Good guess: c_ _
----
You have 3 guesses left.
The available letters are: abdefghijklmnopqrstuvwxyz. Guess a letter and press enter: c
Game over :-(. Your score is 0 points.
```

You are now all set! Good luck implementing hangman!

Skip to main content

Play an interactive game of Hangman.

This function should launch an interactive game of Hangman. The details of the game is defined in the project description available online, and should be read carefully.

- The game should first display how many letters are in the secret word. You should start by generating this output.
- Before each round, the user should see how many guesses that are left and which letters are not yet used
- In each round, the user is prompted to input a letter. Use the `input('..')` function for this.
- The user is given feedback based on whether the letter is in the word or not. The program also performs error handling.
- The game terminates when the user win, has exceeded the number of guesses, or if the user makes an illegal input. in this case the user is shown a score.

Parameters:

- **secret_word** (`str`) – The secret word to guess, for instance `"cow"`
- **guesses** (`int`) – The number of available guesses, for instance `6`