Strings in Python

Thomas Schwarz, SJ

Strings

- Basic data type in Python

  - Strings are <u>immutable</u>, meaning they cannot be shared

    - Why?

      - It's complicated, but string literals are very frequent. If strings cannot be changed, then multiple occurrences of the same string in a program can be placed in a single memory location.

      - More importantly, strings can serve keys in key-value pairs.

String Literals

- String literals are defined by using quotation marks

  - Example:
    ```
    >>> astring = "Hello World"
    >>> bstring = 'Hello World'
    >>> astring == bstring
    True
    ```

  - To create strings that span newlines, use the triple quotation mark

  ```
  >>> cstring = """This is a very
  complicated string with a few
  line breaks."""
  >>> cstring
  'This is a very\ncomplicated string with a few\nline breaks.'
  ```

Escapes

- Python is very good at detecting your intentions when processing string literals

  - E.g.: `"It's mine"`

  - Still sometimes need to use the escape character

    - \t, \n, \", \', \\, \r

    - \xhh —> character with hex value 0xhh

  - Python 3 uses machine conventions for endings

- Python 3 uses utf-8 natively

  - `greetings = ("शुभ प्रभात", "सुप्रभात", "शुभ प्रभात")`

Docstrings

- Doc strings

  - String literals that appear as the first line of a module, function, class, method definition

  - All these items should have a docstring

  - The docstring replaces the help string in Idle and IPython/Jupyter

  - Indent them under the indentation of the object they describe

Docstrings

- Always use triple quotation marks

  - Even for one-liners

```python
def is_anagram(string):
    """checks whether a string is the same spelled forward or backward."""
    return string == string[::-1]
```

```
>>> help(is_anagram)
Help on function is_anagram in module __main__:

is_anagram(string)
    checks whether a string is the same spelled forward or backward.
```

Docstrings

- Example

```python
def change_vowels_for_numbers(astring):
    """ takes all the vowels in the input and replaces them with
        numbers:
        a,A --> 1, e,E --> 2, i,I --> 3, o,O --> 4, u,U -->5
    """
    result = []
    for letter in astring:
        if letter in 'aA':
            result.append('1')
        elif letter in 'eE':
            result.append('2')
        elif letter in 'iI':
            result.append('3')
        eli
        eli
        els
    return
```

```
>>> change_vowels_for_numbers("Thomas Johannes Emil Schwarz")
'Th4m1s J4h1nn2s 2m3l Schw1rz'
>>> help(change_vowels_for_numbers)
Help on function change_vowels_for_numbers in module __main__:

change_vowels_for_numbers(astring)
    takes all the vowels in the input and replaces them with
    numbers:
    a,A --> 1, e,E --> 2, i,I --> 3, o,O --> 4, u,U -->5
```

String Methods

- Strings are classes and have many built in methods

  - `s.lower()`, `s.upper()` : returns the lowercase or uppercase version of the string

  - `s.strip()`: returns a string with whitespace removed from the start and end

  - `s.isalpha()` / `s.isdigit()` / `s.isspace()` tests if all the string chars are in the various character classes

  - `s.startswith('other')`, `s.endswith('other')` tests if the string starts or ends with the given other string

String Methods

- There are a number of methods for strings. Most of them are self-explaining

  - `s.find('other')` : searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found

  - `s.replace('old', 'new')`: returns a string where all occurrences of 'old' have been replaced by 'new'

  - `len(s)` returns the length of a string

Strings and Characters

- Python does not have a special type for characters

  - Characters are just strings of length 1.

Accessing Elements of Strings

- We use the bracket notation to gain access to the characters in a string

  - a_string[3] is character number 3, i.e. the <u>fourth</u> character in the string

String Processing

- Since strings are immutable, we process strings by turning them into lists, then processing the list, then making the list into a string.

  - String to list: Just use the list-command

```
>>> a_string = "Milwaukee"
>>> list(a_string)
['M', 'i', 'l', 'w', 'a', 'u', 'k', 'e', 'e']
```

String Processing

- Turn lists into strings with the join-method

  - The join-method has weird syntax

    - `a_string = "".join(a_list)`

      - The method is called on the empty string `""`

      - The sole parameter is a list of characters or strings

  - You can use another string on which to call join

    - This string then becomes the glue

`gluestr.join([str1, str2, str3, str4, str5])`

str1	gluestr	str2	gluestr	str3	gluestr	str4	gluestr	str5

String Processing

- Examples

```
>>> a_list = ['M', 'a', 'h', 'a', 'r', 'a', 's', 'h', 't', 'r', 'a']
>>> "".join(a_list)
'Maharashtra'
>>> " ".join(a_list)
'M a h a r a s h t r a'
>>> "_".join(a_list)
'M_a_h_a_r_a_s_h_t_r_a'
>>> "oho".join(a_list)
'Mohoaohohohoaohorohoaohosohohohotohorohoa'
```

String Processing

- Procedure:

  - Take a string and convert to a list

  - Change the list or create a new list

  - Use join to recreate a new string

- Alternative Procedure:

  - Build a string one by one, using concatenation (+ -operator)

  - Creates lots of temporary strings cluttering up memory

    - Which is bad if you are dealing with large strings.

String Processing

- Example: Given a string, change all vowels to increasing digits.

  - This is used as a (not very secure) password generator

    - Examples:

      - `Wisconsin —> W1sc2ns3n`

      - `AhmedabadGujaratIndia —>`
        `1hm2d3b4dG5j6r7t8nd90`

String Processing

- Implementation:

  - Define an empty list for the result

  - We return the result by changing from list to string

```
def pwd1(string):
    result = [ ]


    return "".join(result)
```

String Processing

- Need to keep a counter for the digits

```
def pwd1(string):
    result = [ ]
    number = 1
```

String Processing

- Now go through the string with a for statement

- Create the list that will be returned converted into a string

```
def pwd1(string):
    result = [ ]
    number = 1
    for character in string:

        #append to result here


    return "".join(result)
```

String Processing

- We either append the letter from the string or we append the current integer, of course cast into a string

```python
def pwd1(string):
    result = [ ]
    number = 1
    for character in string:
        if character not in "aeiouAEIOU":
            result.append(character)
        else:
            result.append(str(number))
            number = (number+1)%10
    return "".join(result)
```

String Processing

- Argot

  - A variation of a language that is not understandable to others

    - E.g. Lufardo — an argot from Buenos Aires that uses words from Italian dialects

      - Invented originally to prevent guards from understanding the inmates

      - Some words are just based on changing words
        - vesre - al reves (backwards)
        - chochamu - vesre for muchacho (chap)
        - lorca - vesre for calor (heat)

String Processing

- Argot

  - Pig Latin

    - Children's language that uses a scheme to change English words

    - Understandable to practitioners, but not to those untrained

String Processing

- Argot:

  - Efe-speech

    - A simple argot from Northern Argentina no longer in use

    - Take a word: "muchacho"

    - Replace each vowel with a vowel-f-vowel combination

    - "Muchacho" becomes Mufuchafachofo

    - "Aires" becomes "Afaifirefes"

String Processing

- Implementing efe-speech

  - Walk through the string, modifying the result list

```
def efe(string):
    result = [ ]
    for character in string:
            result.append(SOMETHING)
    return "".join(result)
```

String Processing

- We need to be careful about capital letters

  - We can use the string method lower

    - Which you find with a www-search

```
def efe(string):
    result = [ ]
    for character in string:

        elif character in "AEIOU":
            result.append(character+'f'+character.lower())

    return "".join(result)
```

String Processing

```python
def efe(string):
    result = [ ]
    for character in string:
        if character in "aeiou":
            result.append(character+'f'+character)
        elif character in "AEIOU":
            result.append(character+'f'+character.lower())
        else:
            result.append(character)
    return "".join(result)
```

String Processing

```
>>> efe("Alejandria")
'Afalefejafandrifiafa'
>>>
```

Try it out:

- Implement pig latin

  - Use wikipedia

  - Use testing

Slices

- We already know two sequence types: lists and strings

  - Sequences can be sliced: A slice is a new object of the same type, consisting of a subsequence

  - Use a bracket cum colon notation to define slices.

  - sequence[a:b] are all elements starting with index a and stoping before index b.
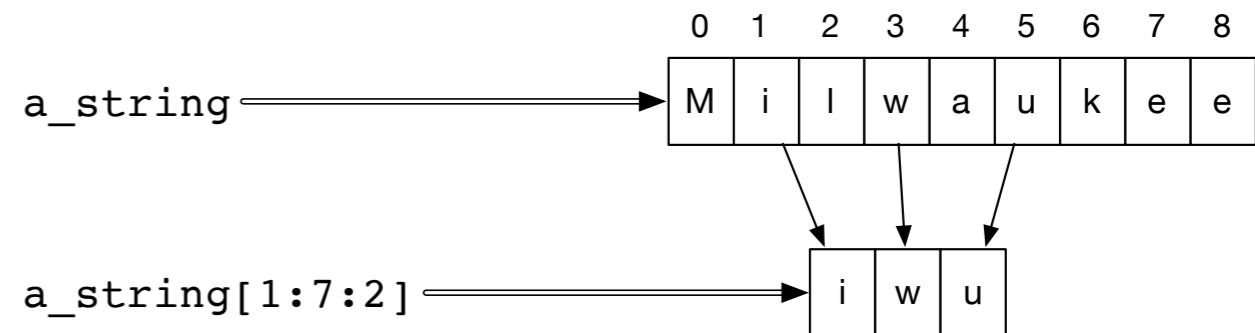
Slices

- String slices
  - Number before colon:
    - Start
  - Number after colon:
    - Stop
  - Default value before colon:
    - Start with first character
  - Default value after colon
    - End with the string

```
>>> a_string = "Milwaukee"
>>> a_string[3:6]
'wau'
>>> a_string[1:5]
'ilwa'
>>> a_string[:6]
'Milwau'
>>> a_string[4:]
'aukee'
```

Slices

- String slices:

  - Optional third parameter is <u>Stride</u>

    - First character is character 1

    - Next one is character 1+2

    - Next one is character 1+2+2

    - Next one would be character 1+2+2+2, but that one is >= the stop value.

```
>>> a_string = "Milwaukee"
>>> a_string[1:7:2]
'iwu'
```

```
      0  1  2  3  4  5  6  7  8
a_string ──────────→ M  i  l  w  a  u  k  e  e

a_string[1:7:2] ──────────→ i  w  u
```

start value is index 1

stop value is index 7

stride is 2

Slices

- Negative strides are allowed.

  - Create a new string that is reversed using default values

```
>>> a_string = "Milwaukee"
>>> b_string = a_string[::-1]
>>> b_string
'eekuawliM'
```

Slices

- Negative strides are allowed

```
>>> a_string = "Ahmedabad, Gujarat, India"
>>> a_string[20:3:-3]
'ItaGda'
```

- Character 20 is "I" of India

- Next character is 17, the "t" in Gujarat

- Stop before character 3 (the fourth character)

Ahmedabad, Gujarat, India

Lists and Strings

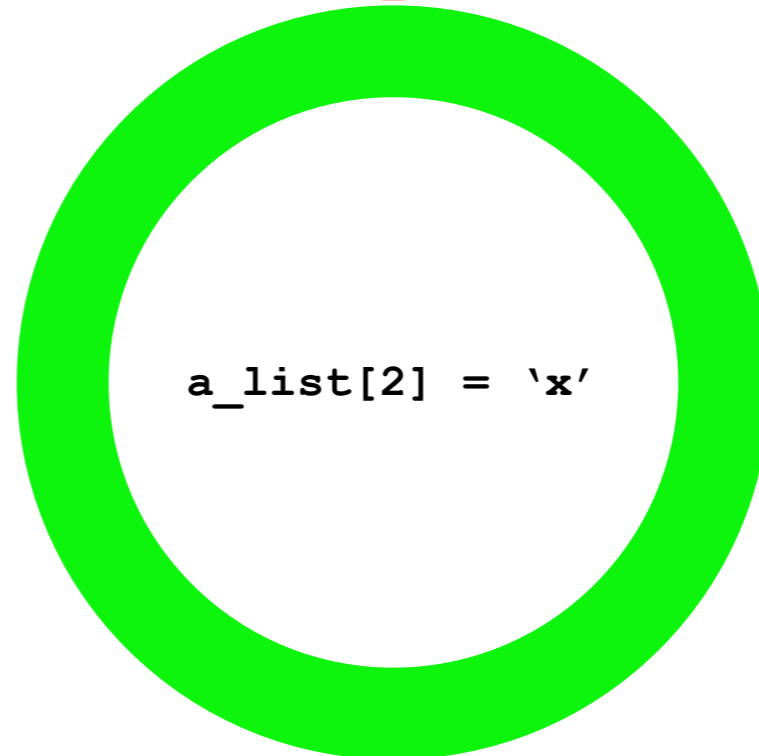- Both lists and strings are sequences

  - Length: `len(a_string)`, `len(a_list)`

  - Concatenation: `a_string + b_string`, `a_list + b_list`

  - Repetition: `3*a_string`, `3*a_list`

  - Membership: `if 'x' in a_string`, `if a in a_list`

  - Iteration: `for ele in a_string`, `for ele in a_list`

Lists and Strings

- Strings are immutable

  🚫 `a_string[2] = 'x'`

- Lists are mutable

  ✅ `a_list[2] = 'x'`
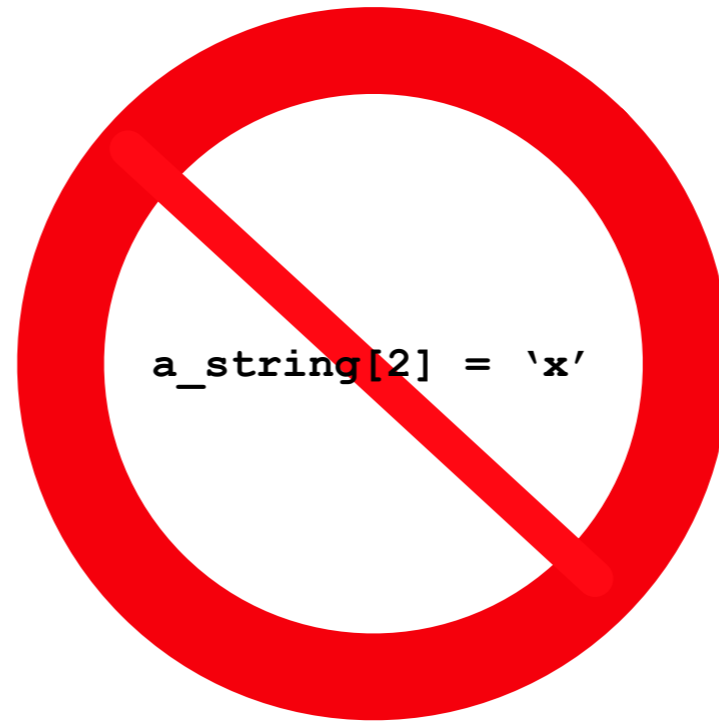
Try it out

- Write a function that determines whether a word is a palindrome (spelled forward the same as backward)

- Write a function that checks whether two words are anagrams (have exactly the same letters).

  - Hint: Without counting letters, you just create an ordered list of the letters in each word

    - For extra credit: remove all non-letters

      - Use string.ascii_letters

Formatting Strings

- We really need to learn how to format strings

  - Python has made several attempts before settling on an efficient syntax.

    - You can find information on the previous solutions on the net.

  - Use the `format` function

    - Distinguish between the **blueprint**

    - and the **string to be formatted**

    - Result is the formatted string.

Formatting Strings

- Blueprint string

  - Uses `{}` to denote places for variables

  - Simple example

    - `"{} {}".format('one', 'two')`

      Blueprint

      Calling format

      String to be formatted

  - Result   `'one two'`

Formatting Strings

- Inside the brackets, we can put indices to select variables

  - 0 means first variable, 1 second, …

  - Can reuse variables

```
>>> "{0}, {0}, {1}, just {0}".format("great", "extraordinary")
'great, great, extraordinary, just great'
```

Formatting Strings

- Additional formatting inside the bracket after a colon

- Can assign the number of characters to print out

```
>>> "{0:10}, {1:10}, {0:10}".format("funny", "nuts")
'funny     , nuts      , funny     '
```

- Default alignment is to the left

Formatting Strings

- Use ^ to center

- Use < to left-align

- Use > to right-align

```
>>> "{0:10}|{1:^10}|{0:>10}".format("sheep", "wolf")
'sheep     |   wolf   |      sheep'
```

Formatting Strings

- Numbers are handled without specifying format instructions.

```
>>> "{} divided by {} is {} modulo {}".format(143, 29, 143//29, 143%29)
'143 divided by 29 is 4 modulo 27'
```

- Or we can insist on special types

  - Use s for string

  - Use d for decimal

  - Use f for floating point

  - Use e for floating point in exponential notation

Formatting Strings

- By specifying "f" we ask for floating point format

- By specifying "e" we ask for scientific format

```
>>> "{0:f}, {0:e}".format(3.141)
'3.141000, 3.141000e+00'
>>>
```

Formatting Strings

- Padding

  - If the variable needs more space to print out, it will be provided automatically

```
>>> "{:10s}".format("Pneumonoultramicroscopicsilicovolcanoconiosis")
'Pneumonoultramicroscopicsilicovolcanoconiosis'
...
```

  - This is actually the longest officially recognized word in English

Formatting Strings

- Padding:

  - On the reverse, we can give the number of significant digits after a period

```
>>> "{:8.2f}".format(3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848)
'    3.14'
```

  - We only want to keep two decimal digits after the period

  - But use a total of 8 spaces for the number.

Formatting Strings

- Escaping curly brackets:

    - If we want to write strings with format containing the curly brackets "{" and "}", we just have to write "{{" and "}}"

    ```
    >>> "{{ {}, {} }}".format(3, 4)
    '{ 3, 4 }'
    ```

    - A single bracket is a placeholder, a double curly bracket is a single one in the resulting string.

Application: Pretty Printing

- Develop a mortgage payment plan

  - Accountants have formulae for that, but it is fun to do it directly

  - Assume you take out a loan of L$ dollars

    - The loan is financed at a rate of r% annually

    - Interest is paid monthly, i.e. at a rate of r/12%

  - Each month you make a repayment

    - Part of the repayment is to pay the interest

    - The remainder pays down the debt

Mortgage Payments

- Use a while-loop
    - Condition is that there is still an outstanding debt
    - Adjust outstanding debt
    - Count the number of payments
- Need to initialize values

Mortgage Payments

- We need values for:

  - Monthly Rate (interest in percent)/1200

  - Principal

  - Repayment

- Get those from the user

  - A true application would contain code that checks whether these numbers make sense.

Mortgage Payments

- Initialization

```
princ = float(input("What is the prinipal "))
rate = float(input("What is the interest rate (in percents)? "))/1200
print("Your minimum rate is ", rate*princ)
paym = float(input("What is the monthly payment? "))
month = 0
```

Mortgage Payments

- We continue until we paid down the principal to zero

```
while princ > 0:
```

Mortgage Payments

- Update the situation in the while loop

- Last payment does not need to be full, so we calculate it

```
intpaid = princ*rate
princ = princ + princ*rate - paym
if princ < 0:
        lastpayment = paym + princ
        princ = 0
month += 1
```

```
*******************************************
**   The Ultimate Mortgage Calculator   **
*******************************************
What is the prinipal 40000
What is the interest rate (in percents)? 4
Your minimum rate is 133.33
What is the monthly payment? 1950
This is what your mortgage scheme looks like
Month     Interest          Principal
---------------------------------
   1        133.33           38183.33
   2        127.28           36360.61
   3        121.20           34531.81
   4        115.11           32696.92
   5        108.99           30855.91
   6        102.85           29008.76
   7         96.70           27155.46
   8         90.52           25295.98
   9         84.32           23430.30
  10         78.10           21558.40
  11         71.86           19680.26
  12         65.60           17795.86
  13         59.32           15905.18
  14         53.02           14008.20
  15         46.69           12104.89
  16         40.35           10195.24
  17         33.98            8279.22
  18         27.60            6356.82
  19         21.19            4428.01
  20         14.76            2492.77
  21          8.31             551.08
  22          1.84               0.00
You paid of the loan in 22 months, and your last payment was 552.92
```

Pretty-Printing Tables

- Format Strings revisited:

  - Format string — blueprint

  - Uses { } to denote spots where variables get inserted

Pretty-Printing Tables

- Syntax

  - {a:^10.3f}

    - a — the number of the variable

      - Can be left out

    - : — what follows is the formatting instruction

    - 10 — number of spaces for the variable

    - . — what follows is the precision

    - 3 — precision

    - f — print in floating point format

Pretty-Printing Tables

- If the variable is larger than the space given:

  - Full value is printed out

  - Alignment by default is

    - left (<) for strings

    - right (>) for numbers

Pretty-Printing Tables

- Task:

  - A program that gives a table for the log and the exponential function between 1 and 10

  - Hint: `x=1+i/10`

```
  x    |      exp(x)      |      log(x)
---------------------------------------------
1.00   |      2.71828     |      0.00000
1.10   |      3.00417     |      0.09531
1.20   |      3.32012     |      0.18232
1.30   |      3.66930     |      0.26236
1.40   |      4.05520     |      0.33647
1.50   |      4.48169     |      0.40547
1.60   |      4.95303     |      0.47000
1.70   |      5.47395     |      0.53063
```

Why another formatting method

- The format method allows very fine-grained control

- But it is verbose

- Python has two type of special strings:

  - r-strings for raw strings: no escapes

  - f-strings for formatting

  - Using f-strings results in more compact and readable code

f-strings

- f-strings are defined with a pair of quotation marks preceded immediately by an "f" or "F"

```
fstring = f'hello world'
```

- An f-string can contain a variable name surrounded by brackets in its definition

- The bracket is then replaced by the value of the variable

f-strings

- Example:

```
number = 6.35
astring = "hello"
fstring = f"{astring}, the number is {number}"
```

- Variable fstring is then

```
'hello, the number is 6.35'
```

f-strings

- The expression in brackets inside an f-string gets evaluated at run time.

- For example, we can say

```
f"{2+3*4}"
```

- or

```
astring = "hello"
string = f"{astring.upper()} World"
```

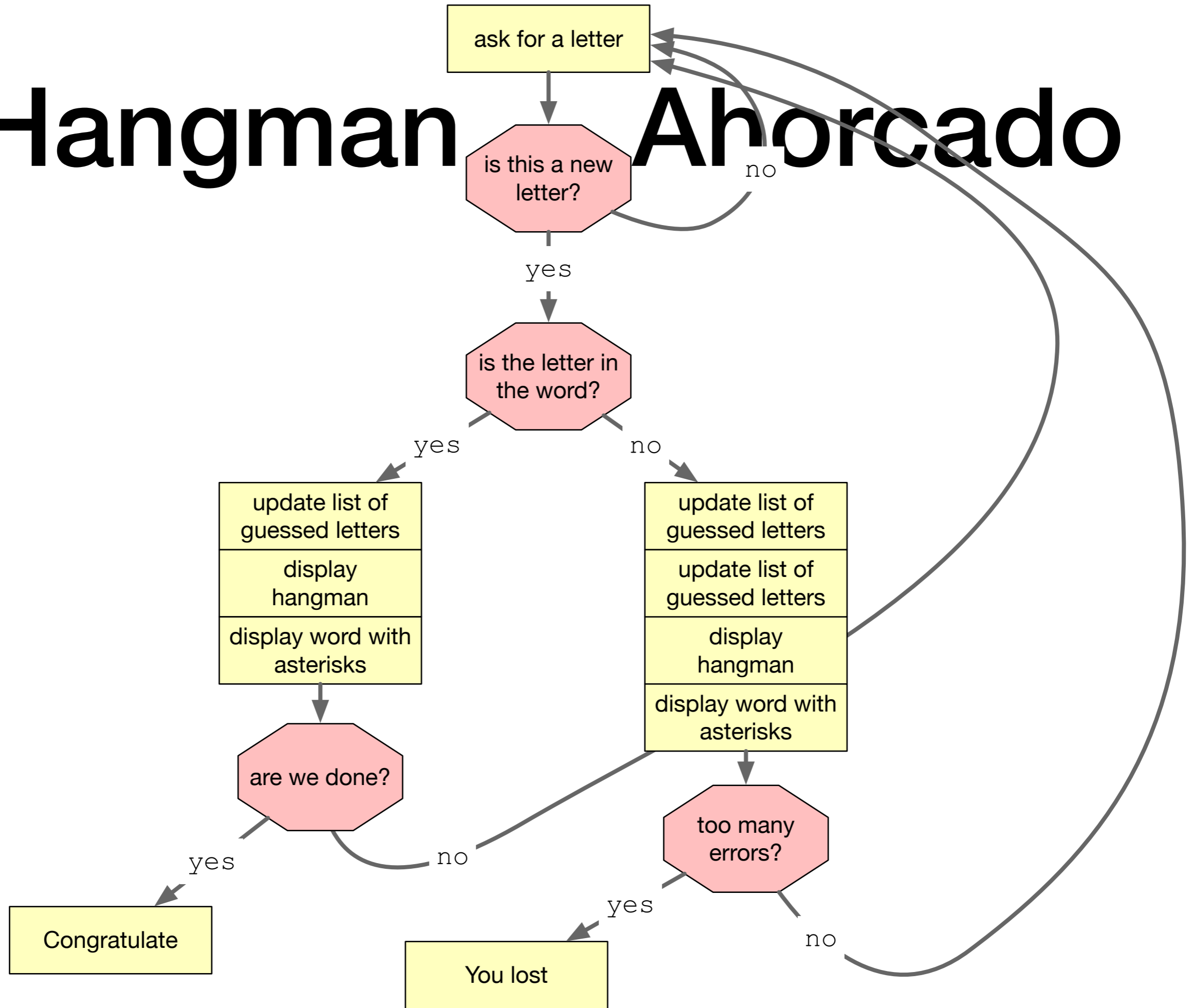which evaluates to

```
'HELLO World'
```

r-strings

- Because of their similarity with f-strings, we mention r-strings

- An r-string uses the escape character only as an escape character, so there is no escaping at all

  - This is useful for strings containing the backslash such as Windows file names

```
address = r"c:\Windows\System32\system.ini"
```

Hangman - Ahorcado

- A slightly morbid childrens' game

  - Guess a word letter by letter

  - For each wrong letter, a part of a hanged man is drawn

```
Enter a letter j
  +------+
  |      |
  |      o
  |     /|\
  |     / \
  |
  |
  |
  |
  |

you looser you
```

Hangman — Ahorcado

- How to plan a software project?

  - Principal idea: divide tasks into simpler components

  - Make a diagram of program logic:

    - This is apt to change

Hangman Ahorcado

```
ask for a letter
        │
        ▼
is this a new letter? ──no──► (back to ask for a letter)
        │
       yes
        │
        ▼
is the letter in the word?
   │              │
  yes            no
   │              │
   ▼              ▼
┌──────────────┐  ┌──────────────┐
│ update list  │  │ update list  │
│ of guessed   │  │ of guessed   │
│ letters      │  │ letters      │
├──────────────┤  ├──────────────┤
│ display      │  │ update list  │
│ hangman      │  │ of guessed   │
├──────────────┤  │ letters      │
│ display word │  ├──────────────┤
│ with         │  │ display      │
│ asterisks    │  │ hangman      │
└──────────────┘  ├──────────────┤
   │              │ display word │
   ▼              │ with         │
are we done?      │ asterisks    │
  │        │      └──────────────┘
 yes      no             │
  │        │             ▼
  ▼        │        too many errors?
Congratulate           │        │
           (no─────►   yes      no
            to are    │        │
            we done)  ▼        (back to
                   You lost     ask for a letter)
```

Hangman — Ahorcado

- Observation:

  - We need a list of guessed letters to decide whether this is a letter

  - We need to do more input control

    - User enters digit

    - user enters capital letters

    - …
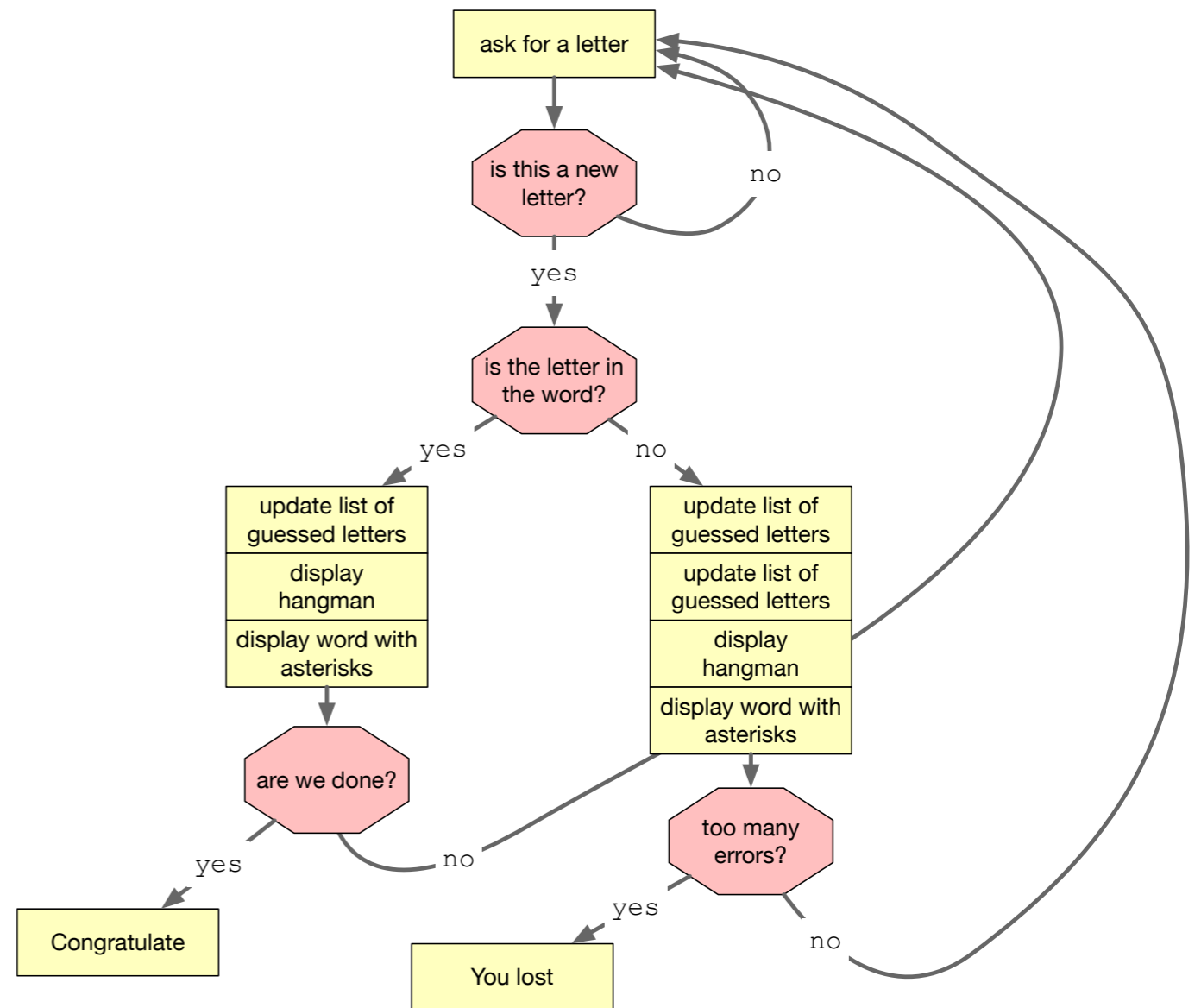
Hangman — Ahorcado

- All of the yellow boxes are candidates for functions

- We can see some common data:

  - The secret word

  - The list of guessed letters

  - The number of bad guesses

ask for a letter

is this a new letter?   no

yes

is the letter in the word?

yes   no

update list of guessed letters

display hangman

display word with asterisks

are we done?

update list of guessed letters

update list of guessed letters

display hangman

display word with asterisks

too many errors?

yes   no

Congratulate

You lost

yes

no

Hangman — Ahorcado

- We can also see that at the heart is a giant loop

- Python-style:

  - Make the loop an infinite loop

  - Break out

Hangman — Ahorcado

- A word about diagrams:

  - Programming has become a lot easier over the years

    - So we program more difficult things

    - And focus has shifted

  - Some methods are very data-centric

    - Useful for big data implementation or graphics, e.g.

  - Some methods focus on processing

    - As we just did

Hangman — Ahorcado

- "Enter a letter" function:

  - Needs one parameter: list of guessed letters

  - Should do error checking (homework / project)

  - Returns a letter not previously seen

```
ask for a
  letter
```

```
Is this a
  new
 letter
```

yes

no

```
 return
 letter
```
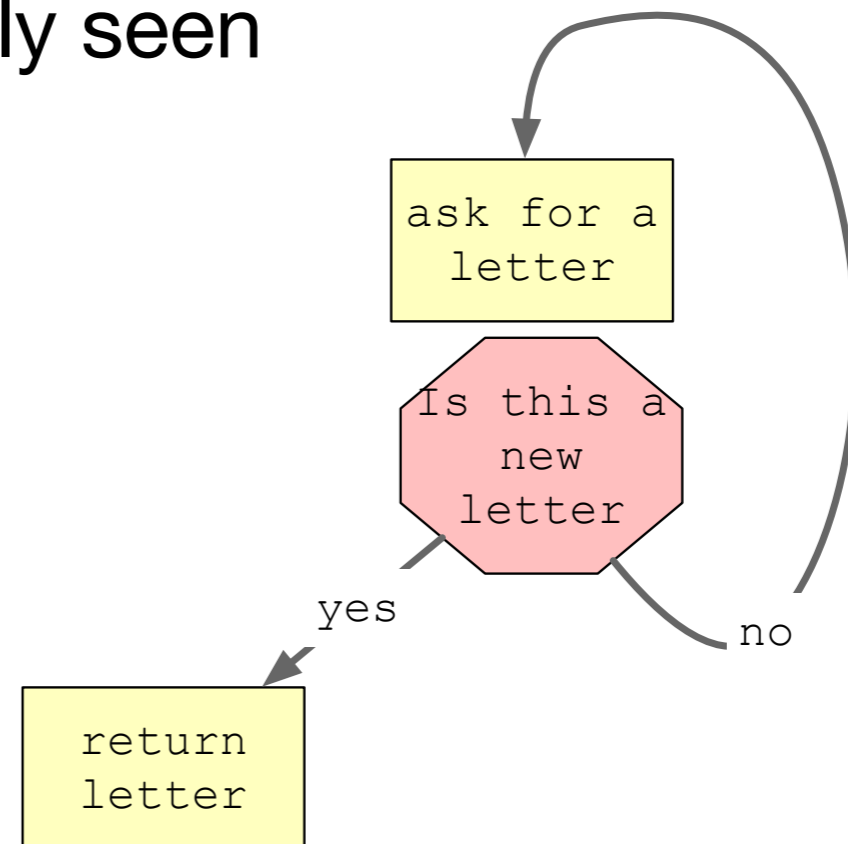
Hangman — Ahorcado

```python
def get_letter(lol):
    while True:
        x = input('Enter a letter ')
        x = x[0]
        if x in lol:
            print('This letter is already guessed. Try again.')
        else:
            return x
```
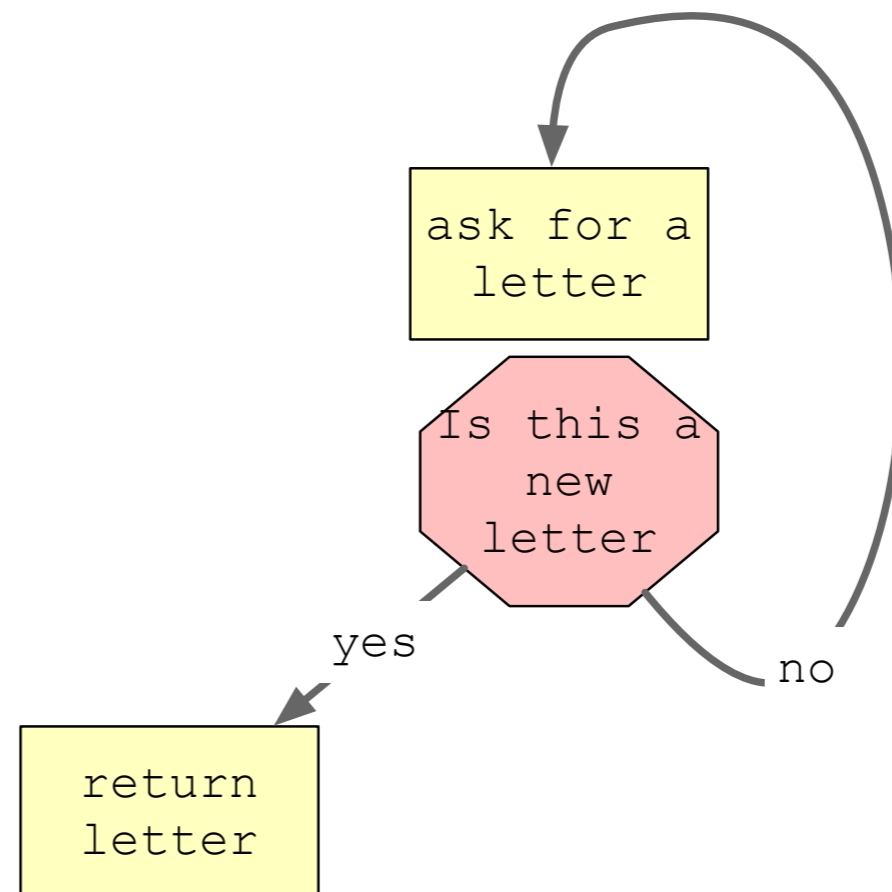
Hangman — Ahorcado

- Check whether we are done

  - All the letters in the secret are in the list of letters already guessed (lol)

```
def done(lol, secret):
    for letter in secret:
        if letter not in lol:
            return False
    return True
```

Hangman — Ahorcado

- Print out the hangman: An exercise in ASCII art

```
Enter a letter a
     +------+
     |      |
     |
     |
     |
     |
     |
     |
     |

Good job. The word is *******a
```

Hangman — Ahorcado
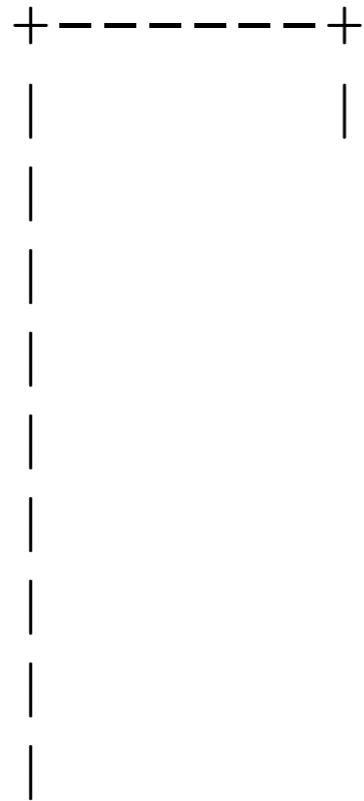
```
Enter a letter b
    +------+
    |      |
    |
    |
    |
    |
    |
    |
    |

Good job. The word is *****b*a
```

Hangman — Ahorcado

```
Enter a letter d
    +------+
    |      |
    |      o
    |
    |
    |
    |
    |

Not quite. The word is c****b*a
```
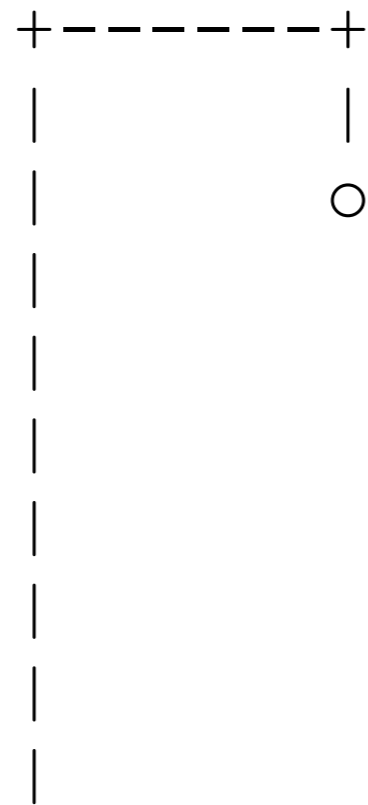
Hangman — Ahorcado

```
Enter a letter e
   +------+
   |      |
   |      o
   |      |
   |
   |
   |
   |
   |

Not quite. The word is c***b*a
```
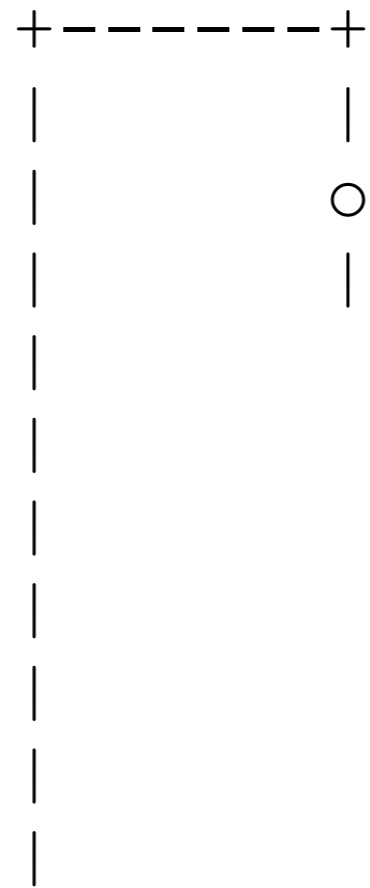
Hangman — Ahorcado

```
Enter a letter f
   +------+
   |      |
   |      o
   |     /|
   |
   |
   |
   |
   |
   |

Not quite. The word is c****b*a
```
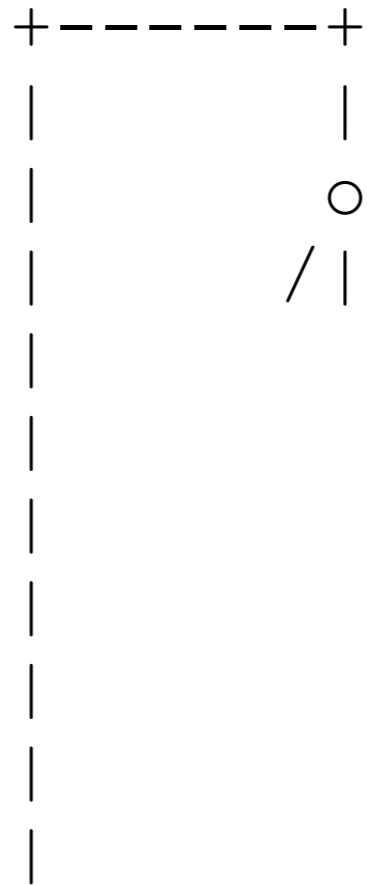
Hangman — Ahorcado

```
Enter a letter g
    +-------+
    |       |
    |       o
    |      /|\
    |
    |
    |
    |
    |
    |

Not quite. The word is c****b*a
```
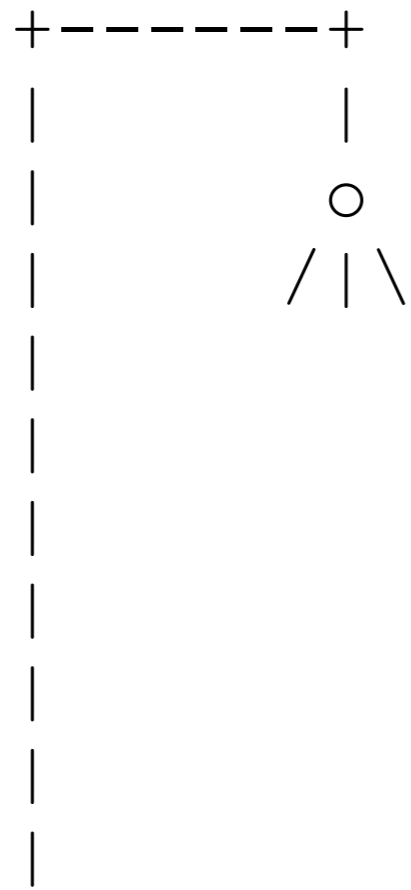
Hangman — Ahorcado

```
Enter a letter h
    +------+
    |      |
    |      o
    |     /|\
    |     /
    |
    |
    |
    |
    |

Not quite. The word is c****b*a
```
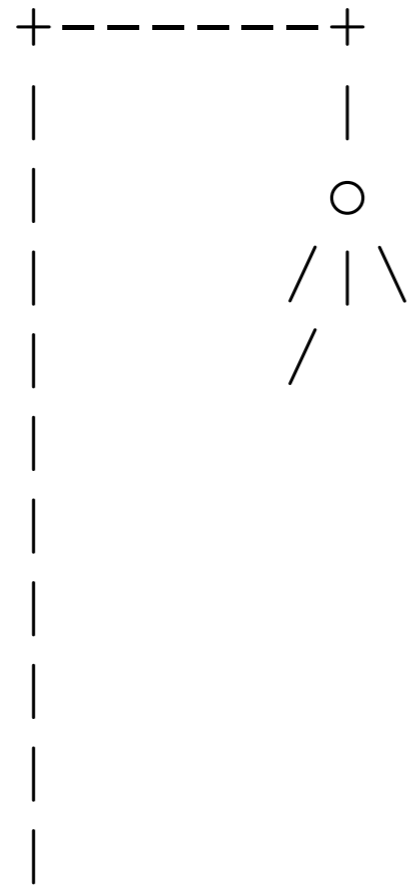
Hangman — Ahorcado
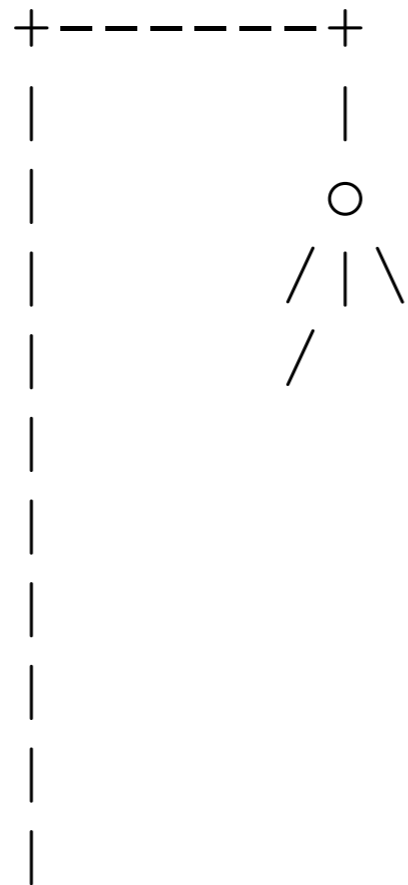
```
Enter a letter i
    +------+
    |      |
    |      o
    |     /|\
    |     /
    |
    |
    |
    |
    |

Good job. The word is c****bia
```

Hangman — Ahorcado

```
Enter a letter j
    +------+
    |      |
    |      o
    |     /|\
    |     / \
    |
    |
    |
    |

  you looser you
```
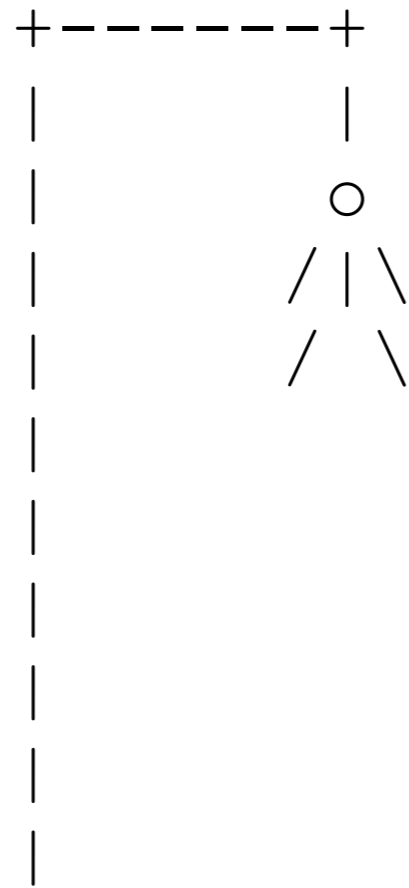
Hangman — Ahorcado

- "printing the hangman"

  - Two possibilities:

    - Draw the same string with slight changes for different number of false guesses

    - Draw different strings (using copy and paste)

  - Can use multi-dimensional strings

    - or use string arithmetic (which becomes unreadable)

```python
def print_it(n):
    if n <= 0:
        print(5*' ' +'+------+\n' + 5*' ' +'|      |\n' + 8*(5*' '+'|\n'))
    elif n == 1:
        print(5*' ' +'+------+\n' + 5*' ' +'|      |\n'
              +5*' '+'|     o\n' +7*(5*' '+'|\n'))
    elif n == 2:
        print(5*' ' +'+------+\n' +  5*' ' +'|      |\n'
              +5*' '+'|     o\n' ++5*' '+'|      |\n'
              +7*(5*' '+'|\n'))
    elif n == 3:
        print(5*' ' +'+------+\n' +  5*' ' +'|      |\n'
              +5*' '+'|     o\n' + 5*' '+'|     /|\n'
              +7*(5*' '+'|\n'))
    elif n == 4:
        print(5*' ' +'+------+\n' +  5*' ' +'|      |\n'
              +5*' '+'|     o\n' + 5*' '+'|     /|\\ \n'
              +7*(5*' '+'|\n'))
    elif n == 5:
        print(5*' ' +'+------+\n' +  5*' ' +'|      |\n'
              +5*' '+'|     o\n' + 5*' '+'|     /|\\ \n'
              +5*' '+'|    /  \n'
              +6*(5*' '+'|\n'))
    elif n == 6:
        print(5*' ' +'+------+\n' +  5*' ' +'|      |\n'
              +5*' '+'|     o\n' + 5*' '+'|     /|\\ \n'
              +5*' '+'|    / \\ \n'
              +6*(5*' '+'|\n'))
```

Hangman — Ahorcado

- Now we are ready for the game:

  - First, define the data structures

```python
def game():
    secret = 'colombia'
    lol = []
    false_guesses = 0
        …
```

Hangman — Ahorcado

- Then start the while loop:

```
def game():
    secret = 'colombia'
    lol = []
    false_guesses = 0
    while True:
        …
```

Hangman — Ahorcado

- First, get the letter and do not forget to update your list of guessed letters (lol)

- We have hidden some logic in get_letter

```
while True:
        x = get_letter(lol)
        lol.append(x)
```

Hangman — Ahorcado

- If the letter is a good guess:

  - Print hangman and word, then check whether we are done

```
if x in secret:
   print_it(false_guesses)
   if done(lol, secret):
       print('You won')
       break
   else:
       print('Good job. The word is', display(secret, lol))
```

Hangman — Ahorcado

- If the letter is bad:

  - update false guesses

  - print hangman

  - decide on whether we lost

```
if x not in secret:
    false_guesses += 1
    print_it(false_guesses)
    if false_guesses >= 6:
        print("you looser you")
        break
    else:
        print('Not quite. The word is', display(secret, lol))
```

Hangman — Ahorcado

- Notice: We could have used return in order to get out of the loop

```python
def game():
    secret = 'colombia'
    lol = []
    false_guesses = 0
    while True:
        x = get_letter(lol)
        lol.append(x)
        if x in secret:
            print_it(false_guesses)
            if done(lol, secret):
                print('You won')
                break
            else:
                print('Good job. The word is', display(secret, lol))
        if x not in secret:
            false_guesses += 1
            print_it(false_guesses)
            if false_guesses >= 6:
                print("you looser you")
                break
            else:
                print('Not quite. The word is', display(secret, lol))
```