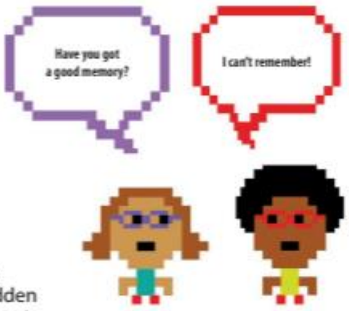
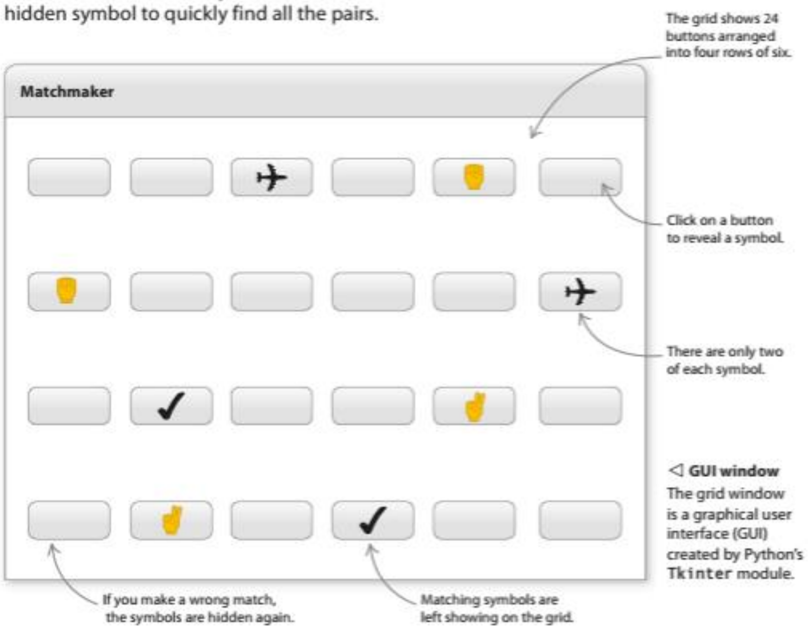
Matchmaker

How good is your memory? Put it to the test in this fun game where you have to find pairs of matching symbols. See how quickly you can find all 12 matching pairs!



What happens

When you run the program, it opens a window showing a grid of buttons. Click on them in pairs to reveal the hidden symbols. If two symbols are the same, you've found a match and the symbols remain visible on the screen. Otherwise, the two buttons are reset. Try to remember the location of each hidden symbol to quickly find all the pairs.


Matchmaker
The grid shows 24 buttons arranged into four rows of six.
Click on a button to reveal a symbol.
There are only two of each symbol.
If you make a wrong match, the symbols are hidden again.
Matching symbols are left showing on the grid.

◁ **GUI window**
The grid window is a graphical user interface (GUI) created by Python's `Tkinter` module.

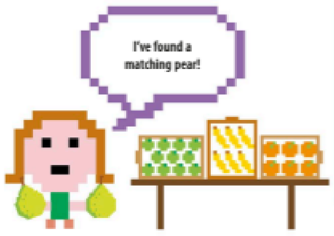
How it works

This project uses the `Tkinter` module to display the button grid. `Tkinter`'s `mainloop()` function listens for button presses and handles them with a special kind of function, called a `lambda` function, that reveals a symbol. If an unmatched symbol has already been revealed, the program checks to see if the second one matches. The project stores the buttons in a dictionary and the symbols in a list.

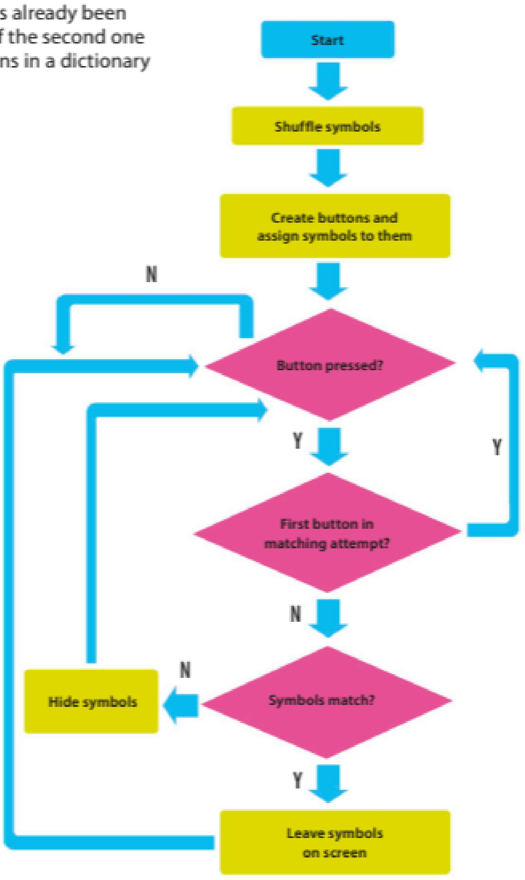
EXPERT TIPS

Lambda functions

Like `def`, the keyword `lambda` is used to define functions. `Lambda` functions are all written on one line and can be used anywhere you need a function. For example, the function `lambda x: x*2` doubles a number. You can assign it to a variable, such as `double = lambda x: x*2`. Then you call it using `double(x)`, where x is a number. So `double(2)` would return 4. `Lambda` functions are very useful in GUI programming, where several buttons may need to call the same function using different parameters. Without the `lambda` functions in Matchmaker, you would have to create a different function for each button – that's 24 functions!



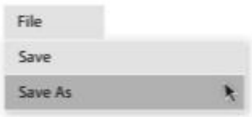
▽ Matchmaker flowchart
After shuffling the symbols and creating the grid, the program spends its time listening for button presses. It ends when all the matching pairs have been found.



Getting started

In the first part of the project, you'll set up the graphical user interface (GUI) and add the pairs of symbols that will be hidden by the buttons.

1 Create a new file
Open IDLE. Create a new file and save it as "matchmaker.py".



2 Add modules
Now type this code at the top of your file to import the modules you need for this project. You'll use `random` to shuffle the symbols, `time` to pause the program, and `Tkinter` to create the GUI.

```
import random
import time
from tkinter import Tk, Button, DISABLED
```

`DISABLED` stops a button from responding after its symbol has been matched.

`Button` creates the buttons in the `Tkinter` window.

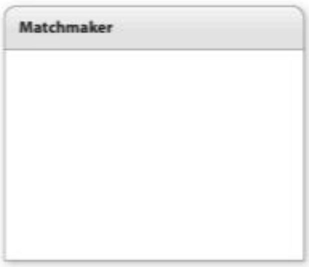
3 Set up the GUI
Under the import commands, add this code, which will set up the GUI. The `root.resizable()` function prevents the player from resizing the window. This is important, since changing the size of the window will mess up the button layout that you'll create later on.

```
root = Tk()
root.title('Matchmaker')
root.resizable(width=False, height=False)
```

These lines create a `Tkinter` window and give it a title.

This line keeps the window at its original size.

4 Test your code
Now run the code. You should see an empty `Tkinter` window with the heading "Matchmaker". If you can't see it, it's probably hidden behind other windows.



Don't forget to save your work.

5 Make some variables

Under the code for Step 3, add the variables that the program needs, and create a dictionary to store the buttons in. For each attempt at a match, you need to remember whether it's the first or second symbol in the match. You also need to keep track of the first button press so you can compare it with the second button press.

```
root.resizable(width=False, height=False)

buttons = {}
first = True
previousX = 0
previousY = 0
```

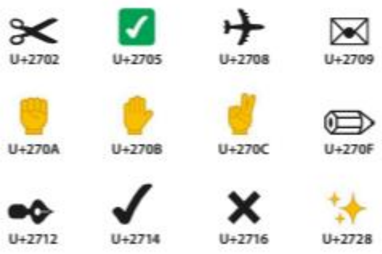
This is the dictionary.

This variable is used to check if the symbol is the first in the match.

These two variables keep track of the last button pressed.

6 Add the symbols

Next type the code below to add the symbols the game will use. As in the Nine Lives project, the program uses Unicode characters. There are 12 pairs, making 24 in total. Add this code under the variables added in Step 5.



```
previousY = 0

button_symbols = {}
symbols = [u'\u2702', u'\u2702', u'\u2705', u'\u2705', u'\u2708', u'\u2708',
           u'\u2709', u'\u2709', u'\u270A', u'\u270A', u'\u270B', u'\u270B',
           u'\u270C', u'\u270C', u'\u270F', u'\u270F', u'\u2712', u'\u2712',
           u'\u2714', u'\u2714', u'\u2716', u'\u2716', u'\u2728', u'\u2728']
```

The symbol for each button is stored in this dictionary.

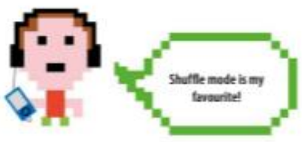
This list stores the 12 pairs of symbols that will be used in the game.

The `shuffle()` function from the `random` module mixes up the shapes.

7 Shuffle the symbols

You don't want the symbols to appear in the same place every time. After several games, the player would remember their positions and would be able to match them all at first attempt, every time. To prevent this, you need to shuffle the symbols before each game starts. Add this line after the list of symbols.

```
random.shuffle(symbols)
```



Bring on the buttons!

In the next stage you'll make the buttons and add them to the GUI. Then you'll create a function called **show_symbol()** to control what happens when a player clicks on the buttons.

EXPERT TIPS

Button

Tkinter has a built-in widget called **Button**, which we use to create the GUI buttons. You can pass different parameters to it. The ones we need are **command**, **width**, and **height**. The **command** parameter tells the program what to do when a button is pressed. This is a function call. In our program, it calls a **lambda** function. The **width** and **height** parameters are used to set the size of the button.

8 Build the grid

The grid will consist of 24 buttons arranged into four rows of six. To lay out the grid, you'll use nested loops. The outer x loop will work from left to right across the six columns, while the inner y loop will work from top to bottom down each column. Once the loops have run, each button will have been given a pair of x and y coordinates that set its position on the grid. Put this block of code after the shuffle command.

```
random.shuffle(symbols)

for x in range(6):
    for y in range(4):
        button = Button(command=lambda x=x, y=y: show_symbol(x, y), \
                        width=3, height=3)
        button.grid(column=x, row=y)
        buttons[x, y] = button
        button_symbols[x, y] = symbols.pop()
```

These are nested loops.

This line creates each button and sets its size and action when pressed.

The button is placed on the GUI.

Use a backslash character if you need to split a long line of code over two lines.

This line saves each button in the **buttons** dictionary.

The button's symbol is set by this line.

△ **How it works**

Each time the loop runs, the **lambda** function saves the current button's x and y values (the row and column it's in). When the button is pressed, it calls the **show_symbol()** function (which you'll create later) with these values, so the function knows which button has been pressed and which symbol to reveal.

REMEMBER

Nested loops

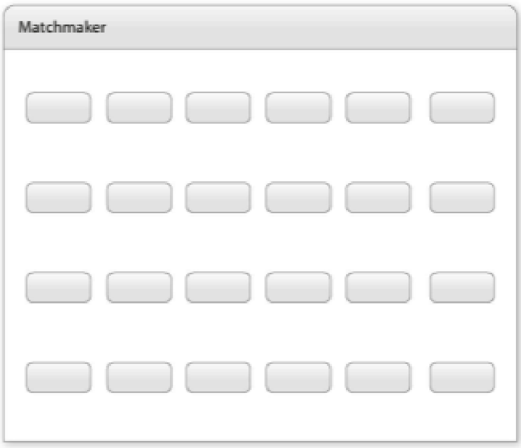
You might remember reading about nested loops on page 35. You can put as many loops inside one another as you want. In this project, the outer loop runs six times. Each time the outer loop runs, the inner loop runs four times. So in total, the inner loop runs 6 x 4 = 24 times.

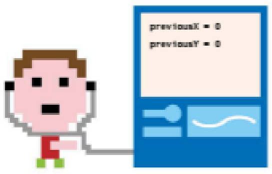
9 Start the main loop
Now start `Tkinter`'s `mainloop`. Once this loop starts, the GUI will be displayed and it will start listening for button presses. Type this line after the code you added in Step 8.

```
        button_symbols[x, y] = symbols.pop()

root.mainloop()
```

10 Test your code
Run the program again. Your `Tkinter` window should now be filled with 24 buttons arranged in a grid. If it doesn't look similar to the picture shown here, check your code carefully for any errors.





```
previousX = 0
previousY = 0
```

11 **Show the symbol**

Finally, you need to create the function that handles the button presses. This function will always display a symbol, but how it operates depends on whether it's the first or second turn in the matching attempt. If it's the first turn, the function just needs to remember which button was pressed. If it's the second turn, it needs to check if the symbols match. Symbols that don't match are hidden. Matching symbols are left showing and their buttons are disabled.

```
from tkinter import Tk, Button, DISABLED

def show_symbol(x, y):
    global first
    global previousX, previousY
    buttons[x, y]['text'] = button_symbols[x, y]
    buttons[x, y].update_idletasks()

    if first:
        previousX = x
        previousY = y
        first = False
    elif previousX != x or previousY != y:
        if buttons[previousX, previousY]['text'] != buttons[x, y]['text']:
            time.sleep(0.5)
            buttons[previousX, previousY]['text'] = ''
            buttons[x, y]['text'] = ''
        else:
            buttons[previousX, previousY]['command'] = DISABLED
            buttons[x, y]['command'] = DISABLED
        first = True
```

The x and y values tell the function which button has been pressed.

These lines tell the program that the variables are global.

These lines show the symbol.

If it's the first turn, the code remembers the button press by storing the x and y coordinates.

Second turn. This line includes a check to stop the player cheating by pressing every button twice!

If the symbols don't match...

If the symbols match...

Disable the pair of matching buttons so the player can't press them again.

Wait 0.5 seconds to give the player time to see the symbols, then hide them.

This line gets the function ready for the first button press of the next attempt.

△ How it works

The function shows a button's symbol by changing its text label to the Unicode character we randomly assigned to it. We use **update_idletasks()** to tell **Tkinter** to show this symbol right now. If it's the first turn, we just store the button's coordinates in variables. If it's the second turn, we need to check the player isn't trying to cheat by hitting the same button twice. If they aren't, we check if the symbols match. If the symbols don't match, we hide them by setting the text to empty strings; if they do match, we leave them showing but disable the buttons.

Hacks and tweaks

You could adapt this game in many ways. You can show the number of moves taken to finish the game, so the player can try and beat their own score or challenge their friends. You could also add more symbols to make the game harder.

Show the number of moves

At the moment, the player has no way of knowing how well they've done or if they've done any better than their friends. How can we make the game more competitive? Let's add a variable to count how many turns a player takes to finish the game. Then players can compete to see who gets the lowest score.

1 Add a new module

You need to import `Tkinter`'s `messagebox` widget to display the number of moves at the end of the game. In the import line, add the word `messagebox` after `DISABLED`.

```
from tkinter import Tk, Button, DISABLED, messagebox
```

2 Make new variables

You'll have to make two extra variables for this hack. One variable will keep track of the number of moves the player makes, while the other will remember how many pairs they've found. Give them both a starting value of 0. Put these lines below the variable `previousY`.

```
previousY = 0
moves = 0
pairs = 0
```

The player hasn't made any moves yet, or found any pairs, so the values are 0.

3 Declare them global

The `moves` and `pairs` variables are global variables, and they'll need to be changed by the `show_symbol ()` function. Let `show_symbol ()` know this by putting these two lines near the top of the function.

```
def show_symbol(x, y):
    global first
    global previousX, previousY
    global moves
    global pairs
```

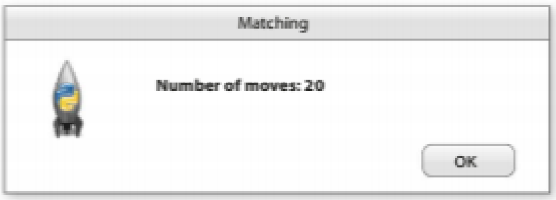
4 **Count the moves**

A move is two button presses (one matching attempt). So you only need to add 1 to the `moves` variable when the `show_symbol()` function is called for the first or the second button press – not for both. Let's do it for the first button press. Change the `show_symbol()` function to look like this.

```
if first:
    previousX = x
    previousY = y
    first = False
    moves = moves + 1
```

5 **Display a message**

Now add the following code near the bottom of the `show_symbol()` function. It will track the matched pairs and show a message box at the end of the game telling the player how many moves they took. When the player clicks the box's OK button, the code calls the `close_window()` function, which we'll add next.



Number of moves: 20

```
    buttons[x, y]['command'] = DISABLED
    pairs = pairs + 1
    if pairs == len(buttons)/2:
        messagebox.showinfo('Matching', 'Number of moves: ' +
                            str(moves), command=close_window)
```

Add 1 to the number of pairs found.

This line displays a box showing the number of moves.

If all the pairs have been found, run the code under this line.

△ How it works

There are 12 pairs of symbols, so you could simply have typed `pairs == 12` in the hack. However, your code is smarter than this. It calculates the number of pairs by using `pairs == len(buttons)/2`. This allows you to add more buttons to the game without having to update this bit of code.

6 **Close the window**

Finally, you need to create a `close_window()` function, to make the program exit the game when the player clicks the OK button on the "Number of moves" message box. Add this code under the line that imports the modules.

```
def close_window(self):
    root.destroy()
```

This command closes the window.

Add more buttons

Let's really challenge the player's memory by adding more buttons and symbols to the game.

1 Extra symbols

First you need to add more pairs to the symbols list. Include this new line in the code.

✳	✴	❄
U+2733	U+2734	U+2744

```
symbols = [u'\u2702', u'\u2702', u'\u2705', u'\u2705', u'\u2708', u'\u2708',
           u'\u2709', u'\u2709', u'\u270A', u'\u270A', u'\u270B', u'\u270B',
           u'\u270C', u'\u270C', u'\u270F', u'\u270F', u'\u2712', u'\u2712',
           u'\u2714', u'\u2714', u'\u2716', u'\u2716', u'\u2728', u'\u2728',
           u'\u2733', u'\u2733', u'\u2734', u'\u2734', u'\u2744', u'\u2744']
```

Add the three pairs of new symbols to the end of the list.

2 Extra buttons

Now add an extra row of buttons. To do this, you just need to change the y range in the nested loops from 4 to 5, as shown on the right.

```
for x in range(6):
    for y in range(5):
```

This line will now create five rows of buttons instead of four.

3 Even bigger?

You now have a total of 30 buttons. If you want to add more, make sure that the number of extra buttons you add is a multiple of 6 so that you always add complete rows. If you're feeling adventurous, you could experiment with different button layouts by changing the nested loops.

❇	❌	❎	❓	❔
U+2747	U+274C	U+274E	U+2753	U+2754
❕	❗	❤	➕	➖
U+2755	U+2757	U+2764	U+2795	U+2796
➗	➡	➰		
U+2797	U+27A1	U+27B0		