

# Applied Programming I

Programming computers in a nutshell.

# Introduction to programming

## Part 3

```
for object to mirror_mod.mirror_object
operation == "MIRROR_X":
mirror_mod.use_x = True
mirror_mod.use_y = False
mirror_mod.use_z = False
operation == "MIRROR_Y":
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation == "MIRROR_Z":
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True
```

```
@selection at the end -add
mirror_ob.select= 1
mirror_ob.select=1
context.scene.objects.active = mirror_ob
("Selected" + str(modifier.name))
mirror_ob.select = 0
= bpy.context.selected_objects[0]
data.objects[one.name].select
```

```
print("please select exactly one object")
```


```
-- OPERATOR CLASSES --
```

```
bpy.types.Operator):
def execute(self, context):
    context.scene.objects.active = mirror_ob
    mirror_ob.select = 1
    print("X mirror to the selected object.mirror_mirror_x")
    return "FINISHED"
```

# Today's objectives

- Understand some principles of programming languages:
  - Organizing software project code in VS Code
  - Namespaces
  - Variables continued: composite data types
  - Pointer, references and memory
  - Functions continued: call by value/reference

# Organizing software project code



Attention: The term software project depends on the context, it can be the software and its files (as here), or the project of developing software.

- Software programs contain (usually) source code that is distributed over several files.
- They also often use libraries that provide functionality that someone else coded.
- The term software project describes the collection of source code, its configuration and further resources.
- An IDE helps to setup and maintain a software project.
- Source code should be organized and structured in different files.

# Python project in VS Code

EXPLORER

NO FOLDER OPENED

You have not yet opened a folder.

[Open Folder](#)

Opening a folder will close all currently open editors. To keep them open, [add a folder](#) instead.

You can clone a repository locally.

[Clone Repository](#)

To learn more about how to use Git and source control in VS Code [read our docs](#).

You can [open a folder containing a .NET project or solution](#), or create a new .NET project.

[Create .NET Project](#)

2

> OUTLINE

> TIMELINE

Welcome

< Welcome

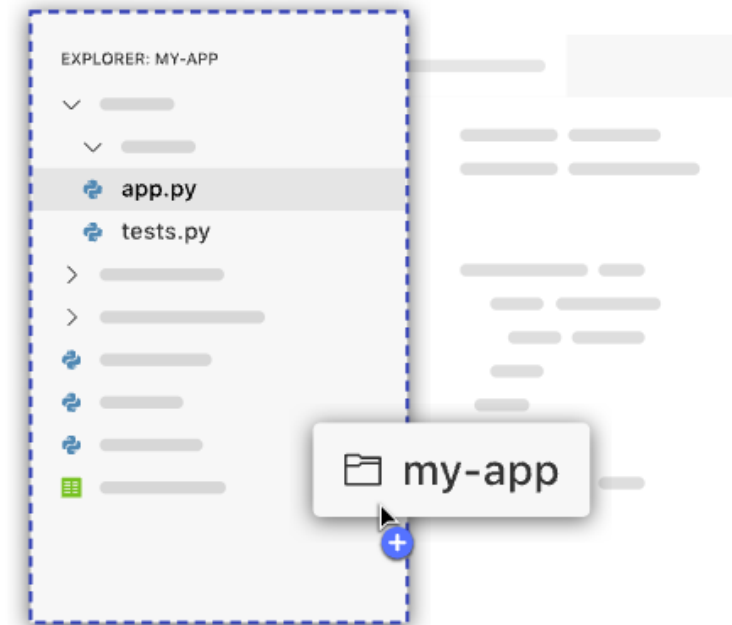
## Get Started with Python Development

Your first steps to set up a Python project with all the powerful tools and features that the Python extension has to offer!

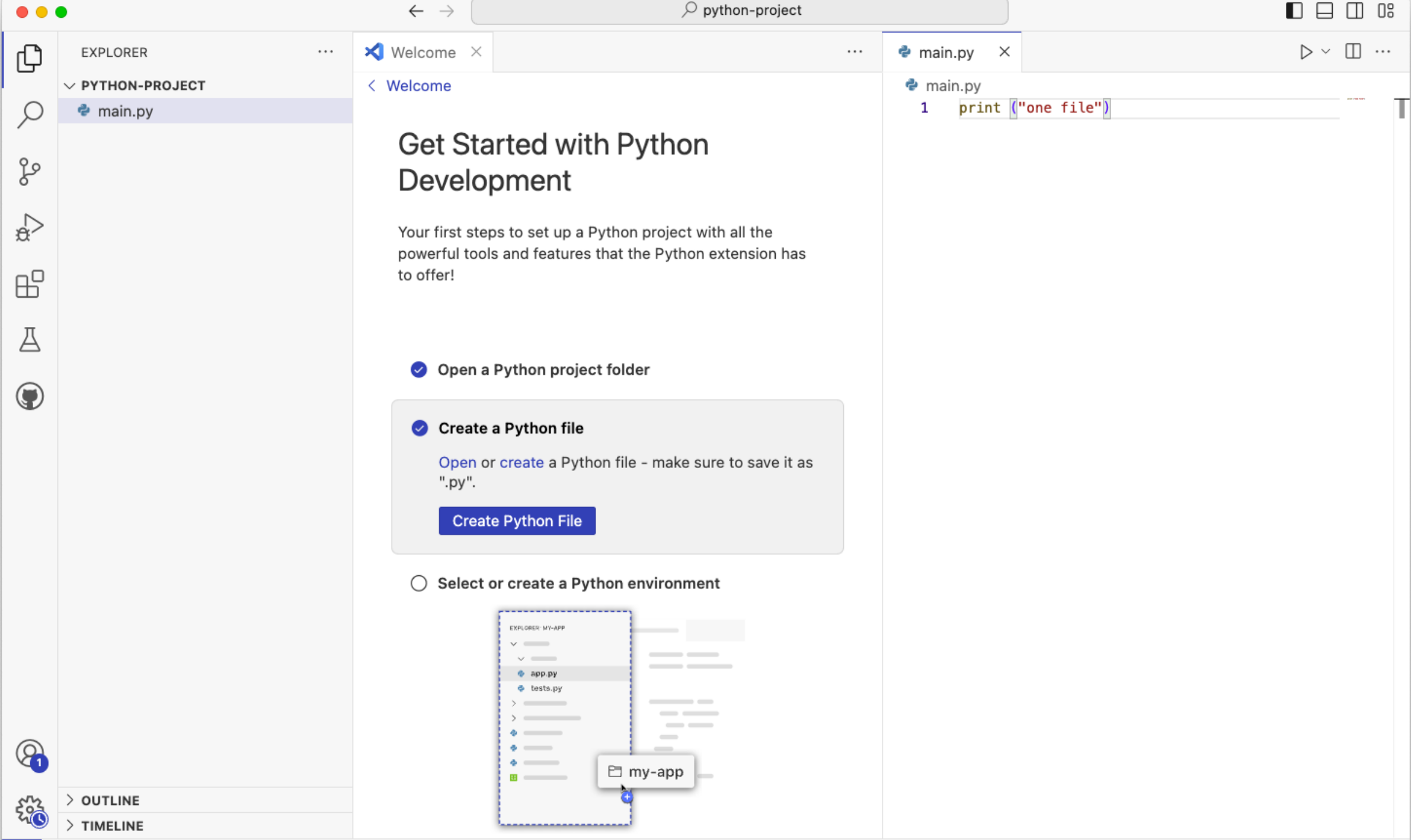
- ☒ **Open a Python project folder**  
[Open](#) or create a project folder.  
[Open Project Folder](#)
- ☐ Create a Python file
- ☐ Select or create a Python environment
- ☐ Run and debug your Python file
- ☐ Keep exploring!

✓ Mark Done

Example in Python









python-project



RUN AND DEBUG



▼ RUN

Run and Debug

To customize Run and Debug  
[create a launch.json file.](#)

[Show all automatic debug  
configurations.](#)

Show automatic Python  
configurations



2



▼ BREAKPOINTS

- ☐ Raised Exceptions
- ☒ Uncaught Exceptions
- ☐ User Uncaught Exceptions

> VENUS OPTIONS



Welcome



## Get Started with Python Development

Your first steps to set up a Python project with all the powerful tools and features that the Python extension has to offer!

✓ Select or create a Python environment

✓ Run and debug your Python file

Open your Python file and click on the play button on



main.py



main.py

```
1 print ("one file")
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS



- (.venv) matthiask@ADM-131916-Mac python-project % /usr/bin/env /Users/matthiask/tmp/vscode/python-project/.venv/bin/python /Users/matthiask/.vscode/extensions/ms-python.debugpy-2024.10.0-darwin-arm64/bundled/libs/debugpy/adapter/../../debugpy/launcher 54000 -- /Users/matthiask/tmp/vscode/python-project/main.py  
one file
- (.venv) matthiask@ADM-131916-Mac python-project %

Venus Terminal

Python Debug Console



Two files. geometry.py is used  
in main.py

Function of Python's module  
math is used.

main.py

main.py

```
1 import geometry
2
3 print(geometry.area_of_circle(4))
4
```

geometry.py

geometry.py > ...

```
1 import math
2
3 def area_of_circle(radius):
4     return math.pi * radius ** 2 # ** is squa
5
6 def circumference_of_circle(radius):
7     return 2 * math.pi * radius
8
9
10
11
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
/Users/matthiask/tmp/vscode/python-project/.venv/bin/python /Users/matthiask/tmp/vsc
ode/python-project/main.py
• (.venv) matthiask@ADM-131916-Mac python-project % /Users/matthiask/tmp/vscode/python
-project/.venv/bin/python /Users/matthiask/tmp/vscode/python-project/main.py
50.26548245743669
○ (.venv) matthiask@ADM-131916-Mac python-project %
```

Venus Terminal  
Python Debug Console  
Python

EXPLORER

PYTHON-PROJECT

> \_\_pycache\_\_

> .venv

geometry.py

main.py

> OUTLINE

> TIMELINE



# Your task:

## Add function for a square for area and perimeter

In groups of two, setup a the Python project and add two functions (area and perimeter) for a square. Time: 20 minutes.

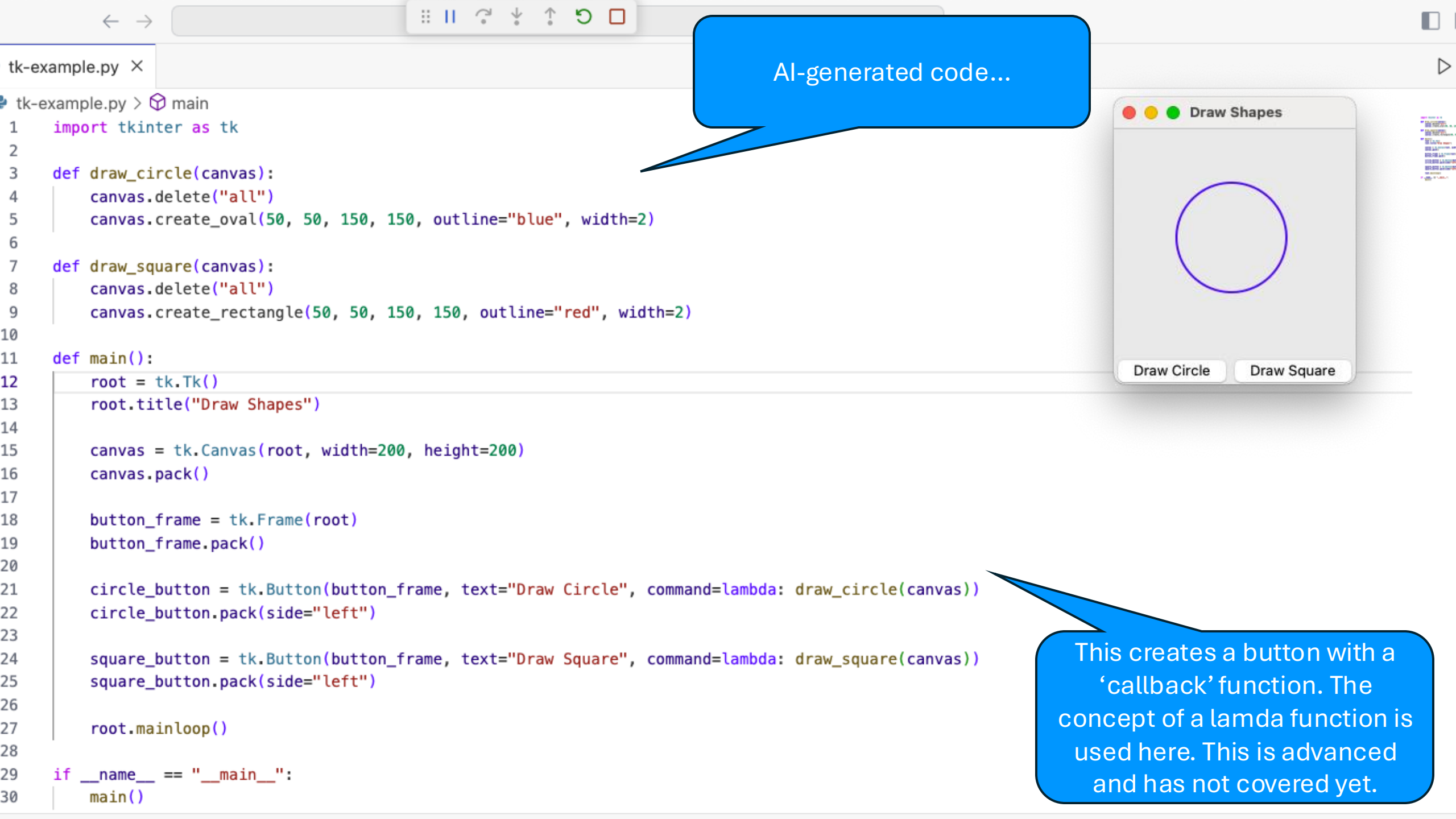
Then, we discuss together.

# Modules/libraries

- Programming languages usually come with core functionality that does (most often) not include further features such as graphical user interfaces (GUI), graphics, networking etc.
- Such features can be included by modules/libraries. The terms are often used interchangeable and may only be distinguished for certain languages.
- The Application Programming Interface (API) and its documentation defines how to use functions of the libraries.

# API example in Python

- Python proposes Tk as default GUI:  
<https://docs.python.org/3/library/tkinter.html>
- This is actually a ‘wrapper’ to the library Tk, e.g., for button:  
[https://www.tcl.tk/man/tcl8.6/TkCmd/ttk\\_button.htm](https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_button.htm)

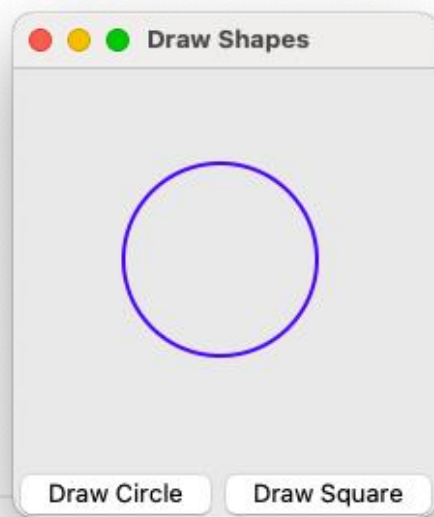


tk-example.py X

tk-example.py > main

```
1  import tkinter as tk
2
3  def draw_circle(canvas):
4      canvas.delete("all")
5      canvas.create_oval(50, 50, 150, 150, outline="blue", width=2)
6
7  def draw_square(canvas):
8      canvas.delete("all")
9      canvas.create_rectangle(50, 50, 150, 150, outline="red", width=2)
10
11 def main():
12     root = tk.Tk()
13     root.title("Draw Shapes")
14
15     canvas = tk.Canvas(root, width=200, height=200)
16     canvas.pack()
17
18     button_frame = tk.Frame(root)
19     button_frame.pack()
20
21     circle_button = tk.Button(button_frame, text="Draw Circle", command=lambda: draw_circle(canvas))
22     circle_button.pack(side="left")
23
24     square_button = tk.Button(button_frame, text="Draw Square", command=lambda: draw_square(canvas))
25     square_button.pack(side="left")
26
27     root.mainloop()
28
29 if __name__ == "__main__":
30     main()
```

AI-generated code...



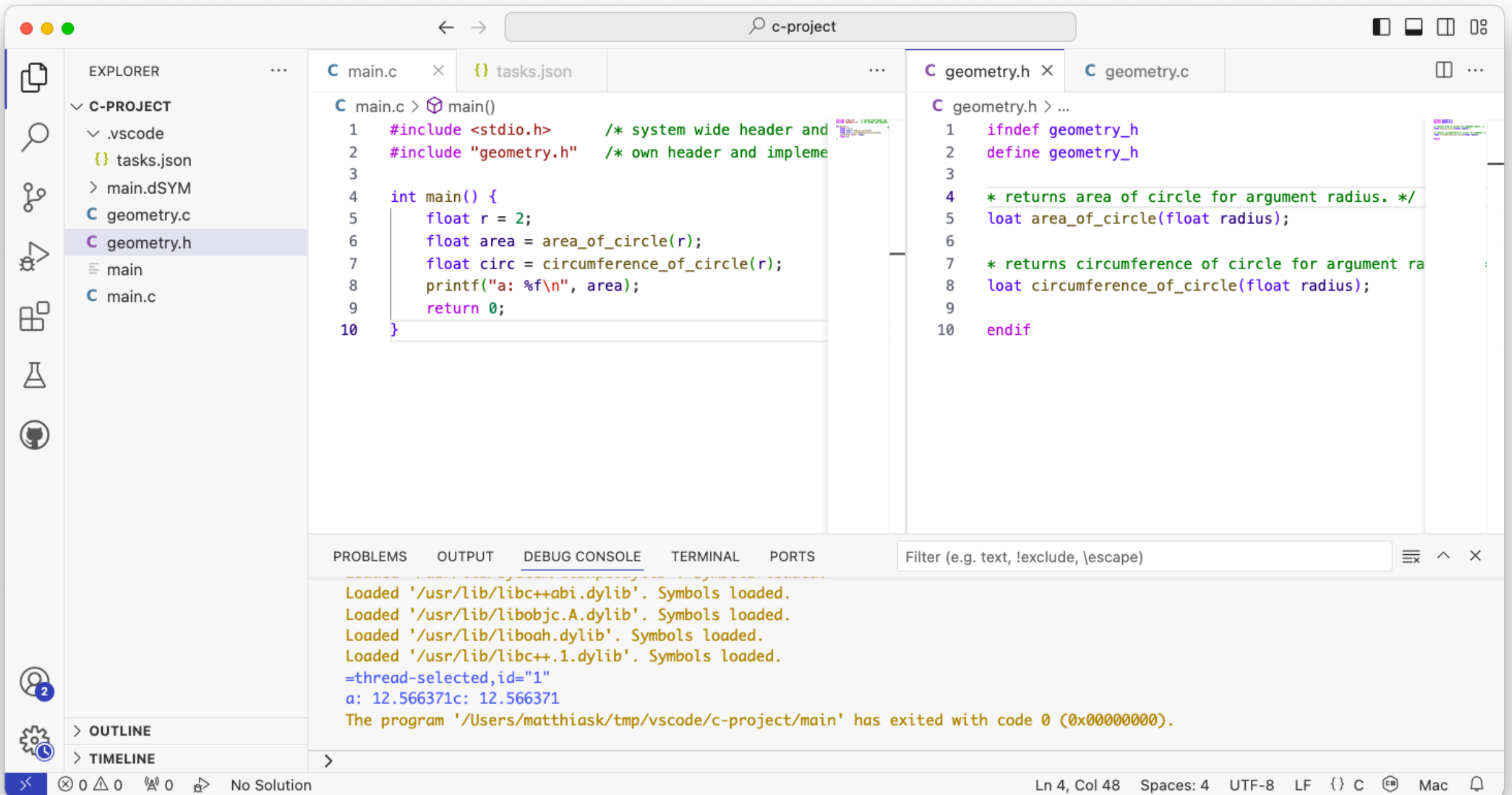
This creates a button with a 'callback' function. The concept of a lambda function is used here. This is advanced and has not covered yet.

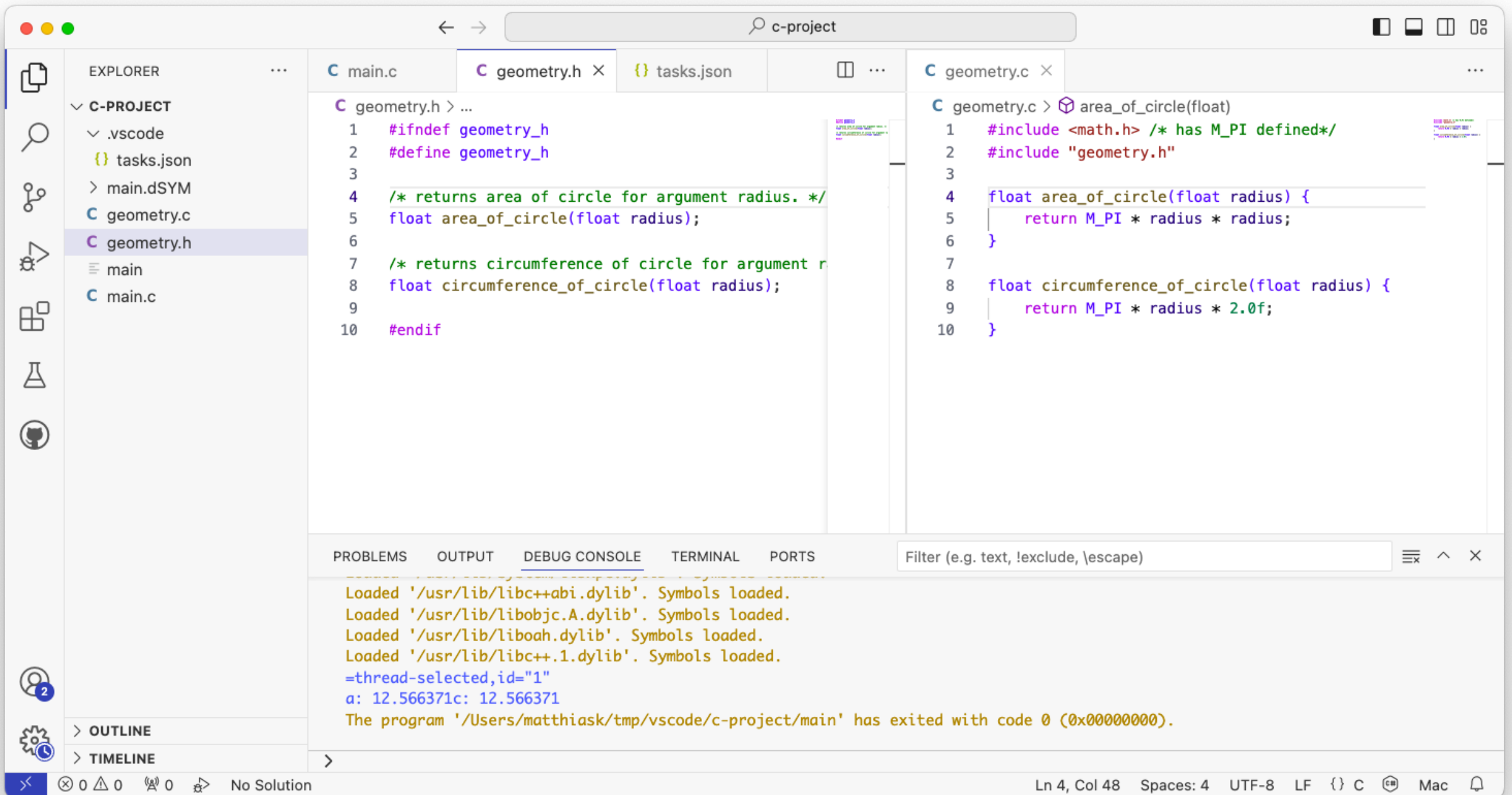
# C project in VS Code

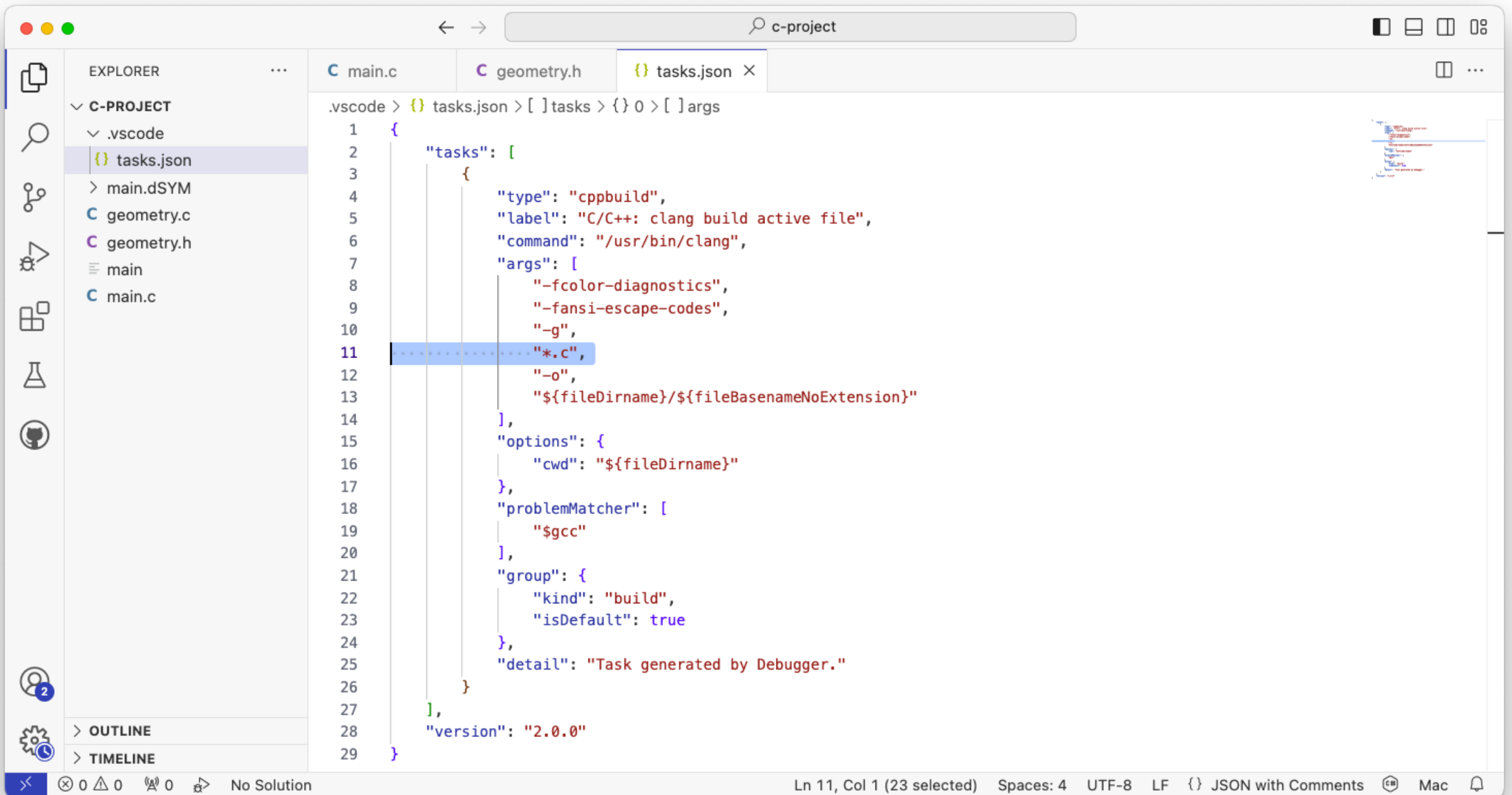


# C projects and C header files

- Including functions from libraries works by including a header in the source for the compile process and linking against binary code that resolves the functions' code.
- Own functions in other files require a header file with the declaration of the functions (how they look like) and a source file (how they are actually written). The source file is compiled and used in the final stage of generating the program.
- To organize this process, there are tools (e.g., make) often provided by the IDE.







# Namespaces



- Working with a large code basis and many libraries can lead to conflicts with variable names but more likely with function names.
- Namespaces provide name scopes that allow to structure identifiers (such as names of variable or functions) in a kind of container. For example, two namespaces can contain a function with the same name, but using the namespace the functions can be distinguished.

# Namespace: examples

- Namespaces do not exist in C.
- Namespaces exist in C# but require object-oriented programming.
- In JavaScript, a namespace can be created with a variable (object).
- In Python, modules and classes (object-oriented concept) can be used as namespaces. See previous example (geometry.py).



# Example of namespace / project in JavaScript

```
// file: geometry.js
```

```
// Object Geometry is also the namespace
```

```
let Geometry = {};
```

```
Geometry.areaOfCircle = function(radius) {  
    return radius * radius * Math.PI; }
```

```
Geometry.circumferenceOfCircle = function(radius) {  
    return 2 * radius * Math.PI; }
```

```
// this is exported to be used elsewhere
```

```
module.exports={Geometry};
```

```
// file: main.js
```

```
// file: include geometry.js
```

```
const { Geometry } = require("./geometry");
```

```
console.log(Geometry.areaOfCircle(4));
```

# Composite data types

- Arrays (fixed and dynamic)
- String (array of characters)
- Structs (own build data type)
- Enumerations
- Pointers and References
- Dictionaries and Sets

# Composite data types



- A composite data type is composed of more than one element of (e.g., primitive) data types.
- A composite data type can contain elements of the same data type, e.g., an array of characters.
- It can also contain elements of different data types, often called struct or record, e.g., a destination given by an area code and a region name.

# Array



- Arrays contain elements of the same data type. An **array** of characters is called **string**.
- Arrays can be of **fixed** length (they can only store a certain number of elements) or **dynamic** (they can 'grow' when storing more elements).
- Elements of an array can usually be accessed by an **index**.

# Array of fixed length

- An array of fixed length is created given the length. Memory is allocated for the array's size only.
- Trying to access elements of the array outside its boundary results in an error. Some languages only have dynamic arrays.
- As example, the following array of characters has a size of 5 elements. The valid range of its indices is 0..4.

| Index   | 0   | 1   | 2   | 3   | 4   |
|---------|-----|-----|-----|-----|-----|
| Element | 'h' | 'e' | 'l' | 'l' | 'o' |

# Array of fixed length example: C

```
char text[5];
```

```
text[0] = 'h';
```

```
text[1] = 'e'; /* [2],[3],[4] are not initialised */
```

```
/* could contain any data */
```

```
int nb[3];
```

```
nb[0] = 1;
```

```
nb[1] = 2;
```

```
nb[2] = nb[0] + nb[1]; /* accessing out of bounds would */
```

```
/* result in an error, (e.g. nb[3]) */
```



# Array of fixed size example: C#

```
int[] numbers = new int[3]; // create array with size 3
numbers[0] = 1;           // assign values
numbers[1] = 2;
numbers[2] = numbers[0]+numbers[1];
// numbers[4] = 0;        //out of bounds error
```

# Arrays of dynamic size

- An array of dynamic size will grow when elements are inserted and shrink when elements are deleted.
- Dynamic arrays and other data structures usually come along with functions used to work with them. These functions are provided by the language or its libraries.
- Such functions with dynamic arrays are e.g., to add an element or to remove an element.
- Some languages do not have a type of dynamic array but have other types such as a list that can be used.

# Array of dynamic size example: JavaScript

```
let numbers = []; // array of any type, empty
numbers.push(3); // add element, pos 0
numbers.push(4); // add element, pos 1
numbers[1] = 2; // overwrite element at pos 1
numbers[8] = 5; // add element at pos 9
console.log(numbers[1]); // ok, element 2
console.log(numbers[3]); // undefined, empty
console.log(numbers[8]); // ok, element 5
```

# Array of dynamic size example: Python

```
numbers = []    # create array
numbers.append(2) # add 2, pos 0
numbers.append(3) # add 3, pos 1
numbers[1] = 4;  # change pos 1 to 4
#numbers[8] = 5; # error pos 8 does not exist
print(numbers[0]) # ok, is 2
print(numbers[1]) # ok, is 4
```



## Your task:

# Write a program with an array of numbers that iterates over the array.

In groups of two, write a program in at one of the languages C, C#, JavaScript or Python. The program shall generate an array with numbers and iterate over all elements of the array to find the largest number. Time: 25 minutes.

# Strings



- A string is an array of characters. Whether the array is fixed or dynamic depends on the the language.
- Some languages, e.g. C, use a string terminator. This is a value that determines the end of the string.
- JavaScript, Python and C# do not use string terminators but they provide a data type string and functions.

The concept of functions will be covered later. For the time being, they can be seen as instructions operating with certain data types.



# Strings example

- Python

```
hi = "hello"
```

- JavaScript

```
let hi = "hello";
```

- C#

```
string hi = "hello";
```

- C

```
char hi[] = "hello";
```

# Structures/Structs

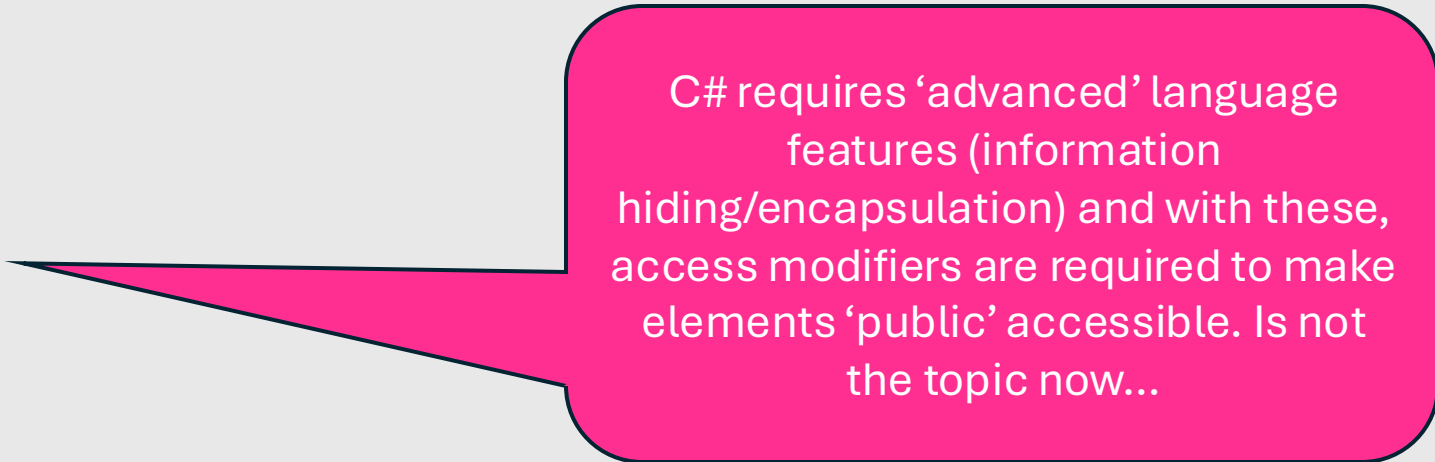


- The composite data type struct (or record) is used to define a structure of elements so that these can be easily accessed. The data types of the elements can be different.
- For example, you can use a struct that defines an address type consisting of the name of a person (type string), a street (type string), a post code (integer) and a city (string).

# Struct example: C#

```
Address address;  
address.name = "S. Error";  
address.street = "Bus 0";  
address.postcode = 6502;  
address.city = "Processor";
```

```
struct Address {  
    public String name;  
    public String street;  
    public int postcode;  
    public String city;  
}
```



C# requires 'advanced' language features (information hiding/encapsulation) and with these, access modifiers are required to make elements 'public' accessible. Is not the topic now...

# Struct example: Python

Python adheres strictly with the object-oriented programming concept. A class is a blueprint of objects. This will covered in later lectures.

```
class Address:
    name = "" # the default values indicate the type
    street = ""
    postcode = 0
    city = ""

address = Address() # struct has the default values initially
address.name = "S. Error"
address.street = "Bus 0"
address.postcode = 6502
address.city = "Processor"

print(address.name)
```

# Struct example: JavaScript

JavaScript is somehow special as it uses its own concepts, function and new will become clear later, hopefully.

```
function Address() { // actually, this is a function that
  this.name = ""; // returns an object (filled struct)
  this.street = ""; // with the default values that also
  this.postcode = 0; // define the type
  this.city = "";
}
```

```
var address = new Address(); // the keyword 'new' does the trick
address.name = "S. Error"; // as it creates the struct address
address.street = "Bus 0";
address.postcode = 6502;
address.city = "Processor";
```

# Struct example: C

C is low-level and memory is managed manually often. The char array is fixed and requires attention and characters are copied.

```
#include <string.h>

struct Address {
    char name[50];          /* attention only space for 50 characters */
    char street[50];
    unsigned int postcode;
    char city[50];
};

/* somewhere else, e.g., in main */
struct Address address;
strcpy(address.name, "S. Error"); /* error prone, second string */
strcpy(address.street, "Bus 0"); /* must have less 50 characters */
strcpy(address.city, "Processor"); /* function copies the characters */
address.postcode = 6502;
```

# Enumerations

- Enumerations are a user-defined type with a set of distinct values, the enumerators (enums).
- An enum are assigned names (mapped to integer values) so that programs are better understandable.
- They are often used in switch-statements.

# Enumeration example: C

/\* type declaration must be before its usage. There is also the keyword typedef which could be used in the declaration but not done here \*/

```
enum color {RED, GREEN, BLUE, YELLOW, WHITE, BLACK};
```

/\* enum indicates the kind of variable. \*/

```
enum color current_color = GREEN;
```

```
switch (current_color) {
```

```
    case BLACK:
```

```
    ...
```



# Enumeration example: C#

```
// current_color is of type Color
```

```
Color current_color = Color.GREEN;
```

```
switch (current_color) {
```

```
    case Color.BLACK:
```

```
        break;
```

```
...
```

```
}
```

```
// type declaration in C# are after its usage on top-level
```

```
enum Color {RED, GREEN, BLUE, YELLOW, WHITE, BLACK};
```

# Enumeration example: JavaScript

// JavaScript needs a detour using JavaScript object data type and keys

```
const Color = { RED : 'red', GREEN : 'green', BLUE : 'blue', YELLOW : 'yellow', WHITE : 'white', BLACK : 'black' };
```

```
let curColor = Color.BLUE;
```

```
switch(curColor) {
```

```
    case Color.RED:
```

```
    ...
```

```
}
```

# Enumeration example: Python

```
from enum import Enum
```

```
class Color(Enum):
```

```
    RED = 1
```

```
    GREEN = 2
```

```
    BLUE = 3
```

```
    YELLOW = 4
```

```
current_color = Color.RED
```

```
match(current_color):
```

```
    case Color.BLUE:
```

```
        print("blue")
```

```
    case _:
```

```
        print("other")
```



# Break

10 minutes

# Pointers and references

Working with memory addresses

# Pointers and references



- **Pointers** and **references** are used to access memory locations of variables.
- Whereas pointers can be uninitialized, references are aliases of variables.
- Using references is safer as using pointers.
- Many languages use references when working with (in particular) composite variables but hidden from the programmer.

# Pointers and references: memory

- Assume there are an integer variable *num* (with value 4) and two reference variables *ref1* and *ref2* that are both refer to *num*.
- Then the memory contains the value 4 at the address (0x1000) of *num*, and this address(0x1000) at the address (0x2000) of *ref1* and at the address (0x2004) of *ref2*.

| Address of memory | Content of memory |
|-------------------|-------------------|
| ...               | ...               |
| 0x1000            | 4                 |
| ...               | ...               |
| 0x2000            | 0x1000            |
| 0x2004            | 0x1000            |
| ...               | ...               |

# Pointer example in C

C uses pointers and not references.

```
int number = 6502;
```

```
int *pointer;           /* the * is for pointer */  
                        /* pointer is not initialised */
```

```
pointer = &number;      /* pointer gets the address, not */  
                        /* the value of number */
```

```
*pointer = 68000;       /* the * dereferences the pointer */  
                        /* change that what is pointed to */  
                        /* number is now 68000 */
```



# Pointers and references in C#

- Pointers can be used as in C (same example as previous).
- References can be used for functions/methods and are used for objects. This will be explained when functions and object-oriented programming is covered.

# Reference example in Python

Python uses references with composite data types but not with primitive data types. It has no pointers.

```
array_1 = [1, 2, 3]    # init the array_1
array_2 = array_1      # array_2 is a reference, not a copy
array_3 = array_1.copy() # this creates a copy
array_2.append(4)      # array_1 and _2 'are' the same
array_1.append(5)      # both references this one
print(array_1)         # output: [1, 2, 3, 4, 5]
print(array_2)         # output: [1, 2, 3, 4, 5]
print(array_3)         # output: [1, 2, 3]
```

# Reference example in JavaScript

As Python, JavaScript uses references with composite data types but not with primitive data types. There are not pointers.

```
let array_1 = [1, 2, 3]; // create array_1
array_2 = array_1;      // array_2 is reference, not copy
array_3 = [...array_1]; // array_3 is a copy (using spread ...)
array_2.push(4);        // array_1 and array_2 reference the same
array_1.push(5);        // both arrays are changed
console.log(array_1);    // output: [ 1, 2, 3, 4, 5 ]
console.log(array_2);    // output: [ 1, 2, 3, 4, 5 ]
console.log(array_3);    // output: [ 1, 2, 3 ]
```

# Memory management and operating system

- Computer programs use memory to store data. To this end, operating systems provide a **memory management** layer. Using system calls (API of the operating system), memory can be allocated and released in applications. There is no operating system on bare metal embedded systems.
- Program code and variables require memory space. An operating system loads the program code (called text) into the memory and prepares further memory space for variables.

# Memory and programming

- Programming languages differ in how they support or hide **memory allocation and deallocation**. Languages such as C require the programmer to think about memory usage. Other languages such as Python use internal mechanisms that hide explicit memory (de)allocation from the programmer.
- Understanding how variables are stored helps for working with, in particular composite, variables and for following concepts.

# Memory: static, stack and heap



Variables are stored in three different categories:

- **Static** memory; storage for global and static variables with two segments: data for initialized variables and bss for uninitialized.
- **Stack** memory: memory segment for call management and local variables. The stack pointer is used and the memory allocation of the stack grows downwards. (Come back to this later...)
- **Heap** memory: memory segment for dynamic memory allocation, starts after the static memory and grows upwards.

Stack (stack memory)  
Grows into direction of heap

Heap (dynamic memory)  
Grows into direction of stack

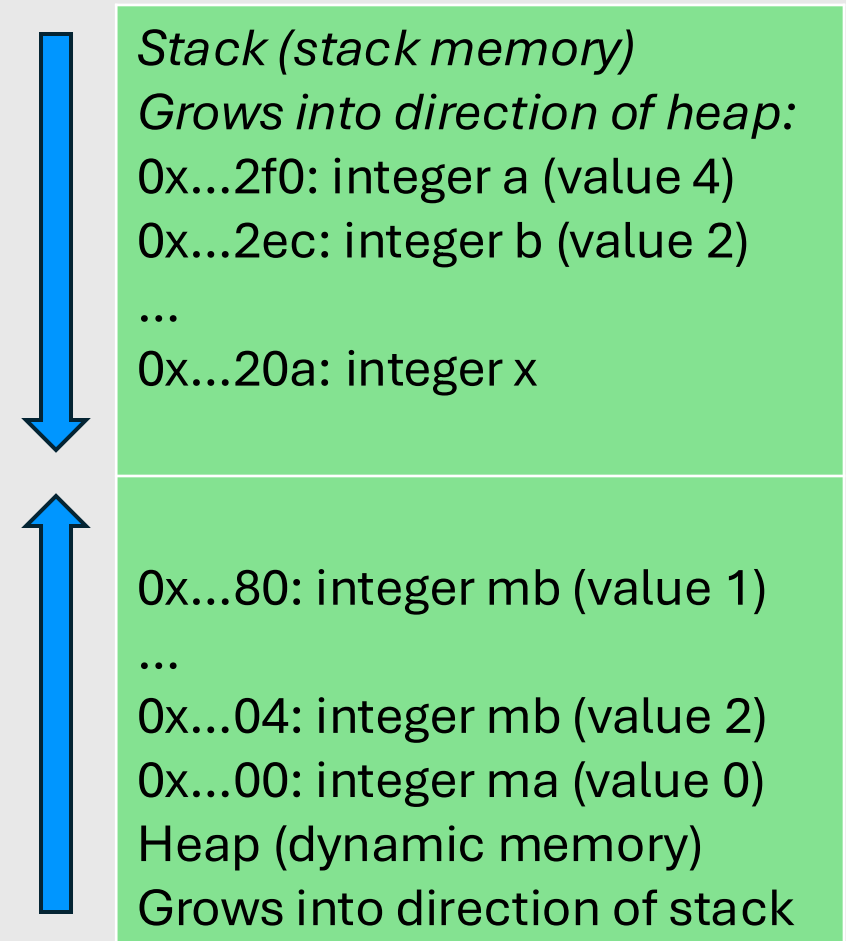
Bss (uninitialized variables)  
Fixed size

Data (initialized variables)  
Fixed size

Text (program code)  
Fixed size

# Overflows

- An overflow happens whether there is no more space available in a memory section.
- Memory sections are using pointers that point to the next available memory location.
- A collision of heap and stack pointer can result memory corruption or end in an security vulnerability.
- Operations systems should avoid this.



# Static and stack memory: illustration in C

- Until now, the example codes did not use the heap but static and stack memory.
- The scope of variables is connected to its memory segment.

```
/* global variable */  
int gx = 0;
```

} Static memory

```
int func(int x) {  
    /* local variable */  
    int fx = x+1;  
    return fx;  
}
```

} Stack memory

```
int main() {  
    /* local variable */  
    int lx = 1;  
    lx = func(lx);  
}
```

} Stack memory



# Heap memory in C

- To use the heap, systems functions need to be used in C.
- The function *malloc* allocates memory. The memory must be released if not longer used by the function *free*.
- The function *malloc* is low-level. It takes as parameter the size of the requested memory in bytes and returns a void pointer (a pointer with no associated type). To request data of specific types, the function *sizeof* should be used and the void pointer should be casted. If something goes wrong, the return value is null.
- The function *free* expects the pointer return by *malloc*.

# Example: malloc and free in C

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    const int nb = 10;
    /* alloc space for nb integers */
    int *p = (int*) malloc(nb*sizeof(int));
    /* pointer p should now have a pointer
    of a space for nb integers */
    /* if p contains 0, something went wrong */
    if (p == NULL) {
        /* exit of program here*/
        return -1;
    }
```

```
    for (int i = 0; i < nb; i++) {
        /* the integer p points to is assigned
        to i and p is incremented */
        *p++ = i;
    }

    for (int i = 0; i < nb; i++) {
        /* decrement p and then dereferenc */
        printf("%d\n", *--p);
    }

    /* p has now its initial value */
    free(p);
    return 0;
}
```

# Garbage collector



- Many programming languages use a **garbage collector** to automatically release memory (on the heap) that is no longer required.
- This means, that the programmer requests memory, e.g. with the keyword *new* in JavaScript and C#, but does not care about releasing the memory.
- Note: in C++, the release of memory is required (*delete*).

# Functions

Part 2

# Functions and call stack



- The stack is managed by the system and using a stack pointer. The **call stack** stores all information about active functions.
- If a function with parameters is called, local variables and the return address of the caller (and sometimes the parameters for the called function) are pushed on the stack. The program counter continues with the called function.
- After the execution of the function, the data is popped off the stack and the program counter gets the return address to continue.
- Thus, a recursive function could lead to a stack overflow.





# Functions and parameters: call by value

- So far, functions using parameter and one return value of primitive data types were shown. The values of these kinds of data types can be easily copied. If the variables are copied from the caller to the callee, the function's parameter are passed **by value**. Example in C#:

```
int f(int x) { // x is a copy of caller's var.  
    x = x+1;  
    return x; }
```

```
int y = 4;  
int z = f(y); // hereafter y = 4 and z = 5
```



# Function and parameters: call by reference

- Parameters of function can also be provided by pointers/ references. Not the value of the variable is copied but the address of the variable. The parameters are passed **by reference**.
- This means, that 'large' variables (e.g. arrays) are not copied (efficient) but that changes of the variables in the function (callee) affect the original variables (caller). Example in C#:

```
int f(ref int x) { // reference: it is the caller's var.  
    x = x+1;      // reference: changes also caller's var.  
    return x; }
```

```
int y = 4;  
int z = f(ref y); // hereafter y = 5 and z = 5
```



# Your task:

## Investigate call by reference and call by value.

In groups of two, investigate how call by reference is possible in Python and JavaScript. Does it work and if so, for which data type? Time: 10 minutes.

# Today's summary

- Organizing software project code in VS Code
- Namespaces
- Variables continued: composite data types
- Pointer, references and memory
- Functions continued: call by value/reference

