Computer Programming Summer Interest Group

Session 7 / Functions

7/28/2025

Exodus 18:13-26

The next day Moses took his seat to serve as judge for the people, and they stood around him from morning till evening. When his father-in-law saw all that Moses was doing for the people, he said, “What is this you are doing for the people? Why do you alone sit as judge, while all these people stand around you from morning till evening?”

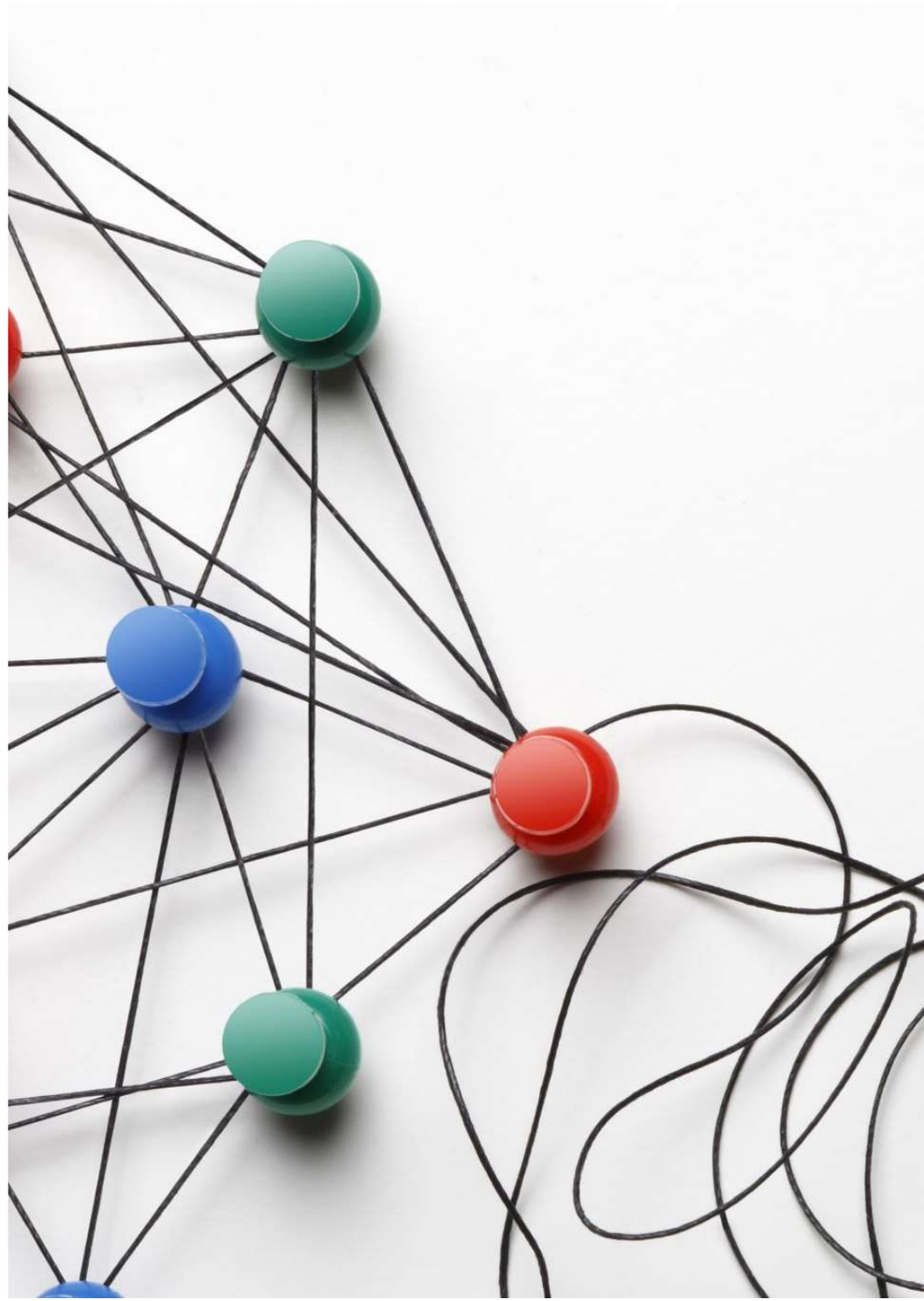
Moses answered him, “Because the people come to me to seek God’s will. Whenever they have a dispute, it is brought to me, and I decide between the parties and inform them of God’s decrees and instructions.”

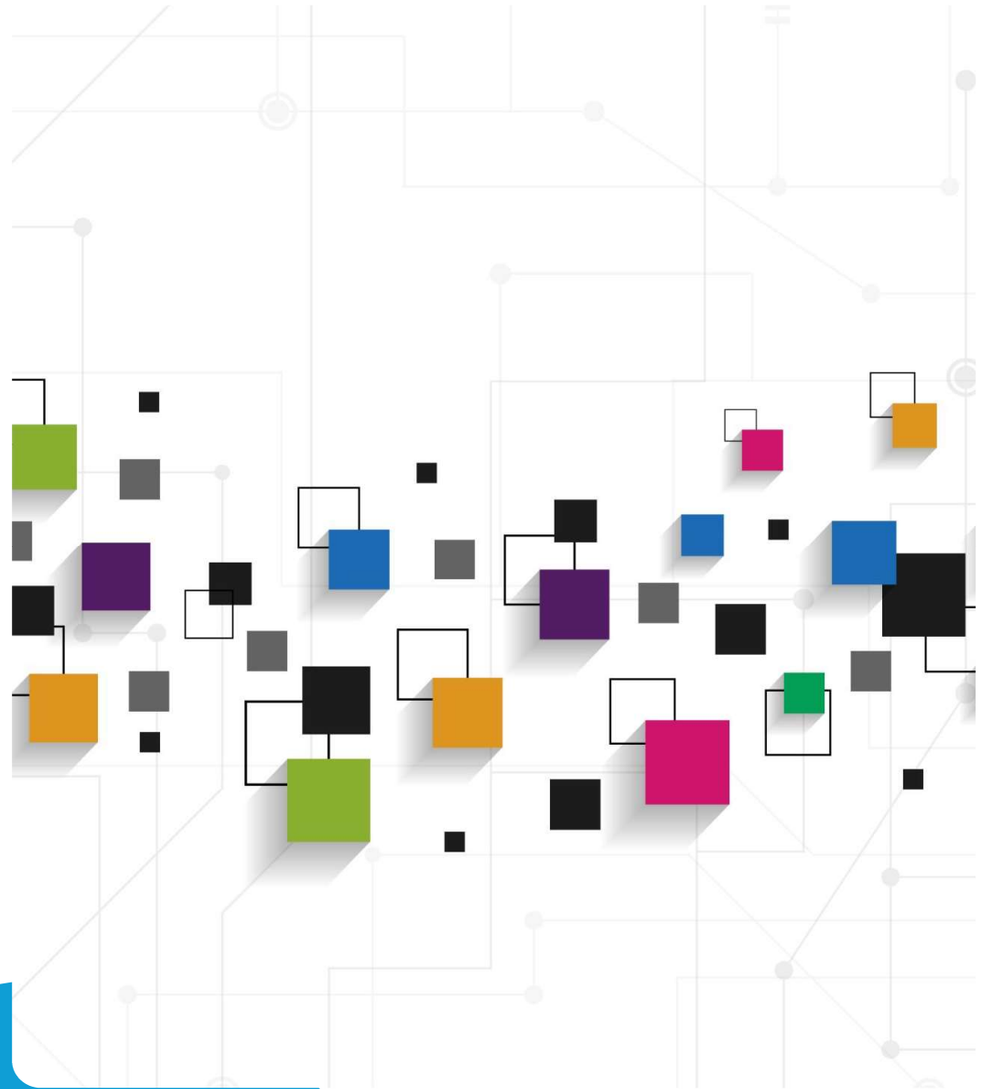
Moses’ father-in-law replied, “What you are doing is not good. You and these people who come to you will only wear yourselves out. The work is too heavy for you; you cannot handle it alone. Listen now to me and I will give you some advice, and may God be with you. You must be the people’s representative before God and bring their disputes to him. 20 Teach them his decrees and instructions, and show them the way they are to live and how they are to behave. But select capable men from all the people—men who fear God, trustworthy men who hate dishonest gain—and appoint them as officials over thousands, hundreds, fifties and tens. Have them serve as judges for the people at all times, but have them bring every difficult case to you; the simple cases they can decide themselves. That will make your load lighter, because they will share it with you. If you do this and God so commands, you will be able to stand the strain, and all these people will go home satisfied.”

Moses listened to his father-in-law and did everything he said. He chose capable men from all Israel and made them leaders of the people, officials over thousands, hundreds, fifties and tens. They served as judges for the people at all times. The difficult cases they brought to Moses, but the simple ones they decided themselves.

Objectives

- By the end of this session, you will be able to:
  - Understand what a function is and why functions are useful.
  - Define and call user-defined functions in Python.
  - Understand the use of parameters and return values.
  - Differentiate between built-in and custom functions.





Functions

- A function is a separate block of code that can be referenced elsewhere in your program
- Functions can be referenced in many places in your code or just a few
- They allow development of modular programs which are easier to write, test and debug

Simple examples

- Functions are specified by using the reserved word def (short for “define”)
- Functions have a name by which they are referred throughout the program
- Functions can take parameters to pass data into the function
- Functions can have “side effects” impacting the state of external variables, objects or devices

```
def greet():
    print("Hello!")

def greet_user(name):
    print(f"Hello, {name}!")



greet()
greet_user(“World”)
```

Returning values from functions

- Functions can return a value using the **`return`** keyword
- The returned value can be assigned to a variable or used directly as an argument for another function

```
def add(a, b):
    return a + b


print(add(3, 5))
print(add(5.2, 4.9))
print(add("Hello, ", "World!"))
print(add(True, False))
print(add([1, 2, 3], [4, 5, 6]))

def generate_greeting(name):
    return f"Hello, {name}!"


print(generate_greeting("World"))
print(generate_greeting(4))
```

Type Hints

- Python allows specification of the intended type of function arguments and return values
- **The Python runtime engine does not enforce these type hints**
- They are useful only to the developer to understand the intent of the function
- Some plugins will highlight to the programmer use of functions which violate the type hints

```
def typed_greeting(name: str) -> str:
    return f"1. Hello, {name}!"

def default_greeting(name: str = "World") -> str:
    return f"2. Hello, {name}!“

print(typed_greeting("World"))
print(default_greeting("Gus"))
print(default_greeting())
```

Passing by reference or value

- All arguments are passed by value … sort of…
- In some cases, the value that is passed is a reference to an object
  - This allows the function to change the value of the referenced object
- In general, complex objects passed into functions have their reference passed, not the value of the object.

```
def double(a):
    a += a
    print(f“inside: {a=}")

x = 3
print(f"before: {x=}")
double(x)
print(f"after: {x=}")

l = [1, 2, 3]
print(f"before: {l=}")
double(l)
print(f"after: {l=}")
```

Some suggestions

- Functions should be well defined
  - Input arguments, return values and side effects should all be well understood.
- Documenting functions is useful to describe it can be useful
  - However, there is a perspective that if the function can only be understood with its documentation, it is too complicated
- Functions should be simple and do only one thing
- Function names and arguments should be descriptive
- These can sometimes be competing issues
  - sum_to_n can be easy to understand, but it is longer and performs poorly especially as n gets larger
  - alt_sum_to_n may be shorter and much faster, but it can be harder to understand.

```
def sum_to_n(n: int) -> int:
    result = 0

    for i in range(n + 1):
        result += i

    return result


def alt_sum_to_n(n: int) -> int:
    return n * (n + 1) // 2


print(sum_to_n(100))
print(alt_sum_to_n(100))
```

Input validation

- Input validation ensures the arguments provided are valid values
- Some convention will be needed to deal with bad values
- For now, return None when a bad value is given

```
def f_to_c(f):
    if not isinstance(f, (float, int)):
        return None

    return (f - 32) * 5 / 9


print(f_to_c(32))
print(f_to_c(212.0))
print(f_to_c(-40.0))
print(f_to_c("Hello, World!"))
```

Recursion

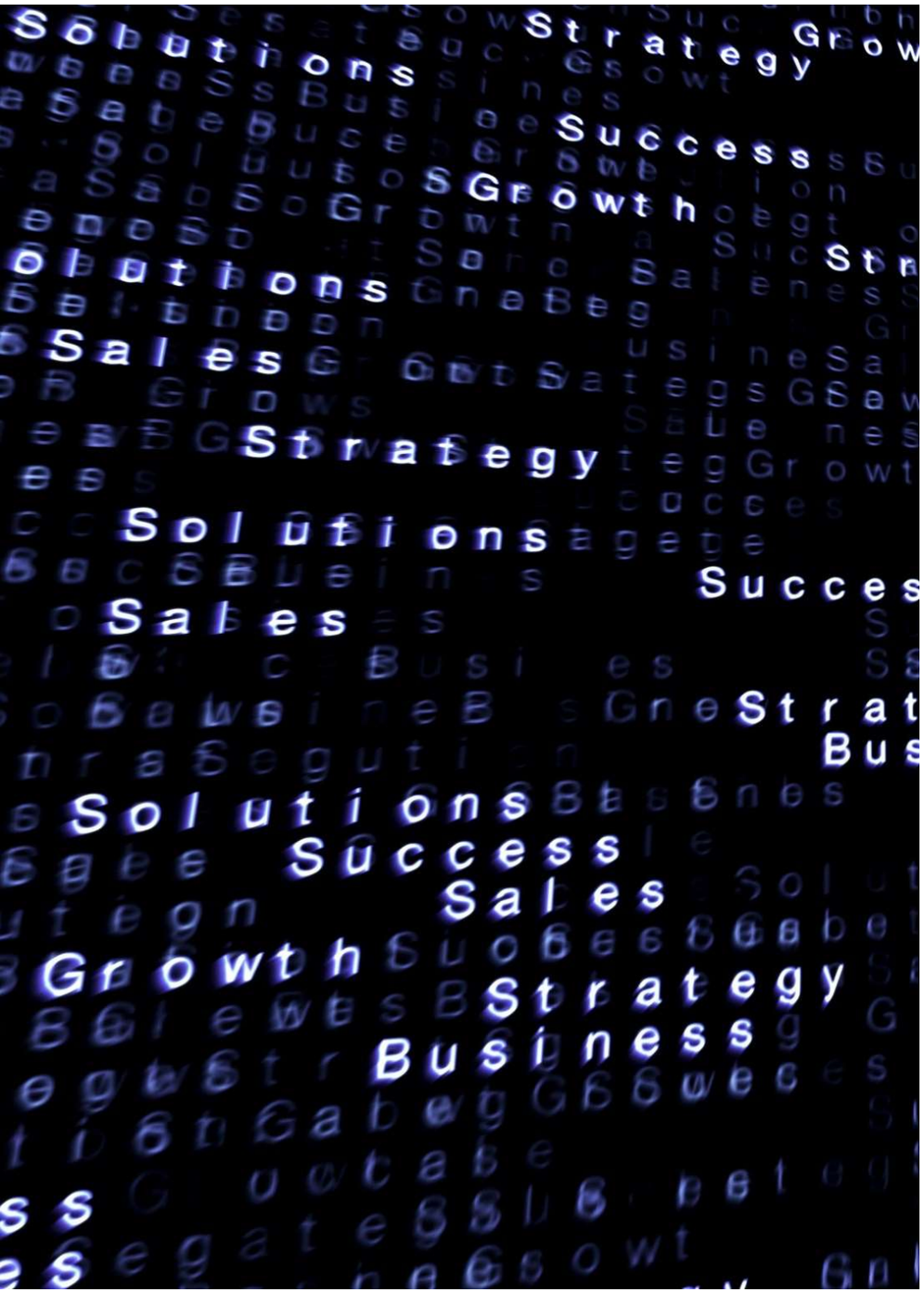
- Recursive functions are those that are defined in terms of themselves
- It will directly or indirectly call itself
- Care must be taken to ensure the recursion ends
- Terminating a recursive function involves checking for some stopping condition
- There are always equivalent iteration-base solutions which can be used instead of recursion
- Recursion generally performs poorly compared to use of loops
- Python limits the depth of recursion

```
def bad_sum_to_n(n: int) -> int:
    return n + bad_sum_to_n(n - 1)


def recursive_sum_to_n(n: int) -> int:
    if n <= 0:
        return 0

    return n + recursive_sum_to_n(n - 1)


print(bad_sum_to_n(100))
print(recursive_sum_to_n(100))
```



Activities

- Write a function that converts Celsius to Fahrenheit.
  - The function should take a single value, the degrees Celsius
  - It should return the equivalent degrees Fahrenheit
  - F = 32 + 9 * C / 5
  - Test with values 0, 100, -40

- Write a function that calculates the factorial of a non-negative number
  - n! = n×(n-1)×(n-2)× … ×2×1
  - If the value is undefined, return None
  - Test with values 5, 10, -1
  - Hint: **`isinstance(x, int)`** will return true if x is an integer value

- Write a function that checks if a given year is a leap year.
  - A leap year occurs when a year is divisible by 4, but not by 100, unless it is also divisible by 400.
  - Test with values 1984, 2025, 2000, 1900, “y2k”