

For loop –

for loops are used to **loop over sequential data** – it considers the first element, then the second element, and so on.

```
tutors = ["Maddy", "Simon", "Ethan", "Maliha", "Sim"]
for tutor in tutors:
    print(f"{tutor} is a COMP1010 tutor")
```

While loop –

while loops are used when you want to **repeat a block of code as long as a condition is true**. Unlike for loops, which run a set number of times, while loops are useful when the **number of repetitions isn't known in advance**.

```
count = 0
while count < 5:
    print(f"Count is: {count}")
    count += 1
```

If-else statement –

if and else statements are used for **conditional logic**, ie telling the computer to make decisions based on values of data.

For example, an if statement could be used when checking a user's age.

```
if user_age < 18:
    print("You're not old enough to enter a bar")
else:
    print("Enjoy your evening!")
```

If-Elif-Else – Multiple conditions:

```
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
else:
    grade = "C"
```

Example A uses **two independent if statements**, hence 'condition1' and 'condition2' will be checked separately. If both conditions are true, both blocks will run.

Example B uses the **if... elif statement**, which means "if the previous conditions were not true, try this condition". If 'condition1' is true, 'condition2' will be skipped. Elif will only run if 'condition1' is false.

List [] –

Lists are used to store **multiple items in a single variable**. These items are ordered and can be accessed by their position (also called index). Lists are useful when you want to **group related values** like names, numbers, or objects.

```
fruits = ["apple", "banana", "mango"]
print(fruits[0]) # prints 'apple'
```

To add data to lists use .append

E.g. data.append(new_player)

Dictionary { } –

Dictionaries store **key-value pairs**, making them ideal for **storing relationships** between different data types. For example, a dictionary can store a user's name, phone number, and address.

```
user_dictionary = {
    'Simon': {
        'phone_number': '0449389278'
        'address': 'Unit 1 Haymarket Street'
    }
    'Lois': {
        'phone_number': '0346179367'
        'address': 'Unit 4 Rhodes Street'
    }
}
print(user_dictionary['Simon']['address'])
```

Dictionaries allow **fast and easy access to data** using unique keys. This makes it quicker to find data without searching through everything.

```
if 'Lois' in user_dictionary:
    print("Lois is in the system.")
```

Nested dictionaries – a dictionary where each value is itself another dictionary. This is useful for **storing grouped information**.

```
students = {
    "Alice": {"major": "Engineering", "grade": 85},
    "Bob": {"major": "Business", "grade": 90}
}
print(students["Alice"]["major"]) # Output: Engineering
```

Accessing info → print(students["Alice"]["major"])

Cookies –

Cookies store data inside the user's browser
Files store data in the server

COOKIES VS JSON ON LAST PAGE

Therefore factors to consider:

Is the data **sensitive** (eg passwords) -- cookies aren't safe, since they are stored on the user's device. Store sensitive data on the server.

Is the data used to **identify a user**? Cookies let us know who a user is, and so we need to store some identifying information in them.

IN WEBPAGE:

SETUP:

```
from flask import Flask, request, session, redirect
app.secret_key = "something_secret" # Required for sessions to work
```

FORM HANDLING + STORING DATA IN COOKIE: # usually do this in /submit route

```
@app.post("/submit")
```

```
def handle_form():
```

```
    # STEP 1: Get existing data from the cookie (or use empty list if it's new)
```

```
    item_list = session.get("item_list", [])
```

```
    # STEP 2: Get new input from the form
```

```
    new_item = {
        "key1": request.form["field1"], # e.g. "colour"
        "key2": request.form["field2"] # e.g. float(request.form["length"])
    }
```

```
    # STEP 3: Add new item to the list
```

```
    item_list.append(new_item)
```

```
    # STEP 4: Save list back into session cookie
```

```
    session["item_list"] = item_list
```

```
    # STEP 5: Redirect to another page to view result
```

```
    return redirect("/preview")
```

PREVIEW PAGE (e.g. /preview) #reads the list from the cookie and displays it.

```
@app.get("/preview")
```

```
def preview():
```

```
    item_list = session.get("item_list", []) # Gets stored cookie list
```

```
    return str(
```

```
        p.html(
```

```
            p.body(
```

```
                p.h1("Preview Items"),
```

```
                p.ul(
```

```
                    *[p.li(f"{item['key1']} - {item['key2']}") for item in item_list]
```

```
                ),
```

```
                p.a(href="/")( "Back" ) # p.a REDIRECTS TO ANOTHER PAGE
```

```
            )
```

```
        )
```

CLEARING THE COOKIE (e.g. after placing an order)

```
@app.post("/order")
```

```
def order():
```

```
    item_list = session.get("item_list", [])
```

```
    # Process the data as needed (e.g. total cost)
```

```
    session.pop("item_list", None) # This clears the cookie list
```

```
    return str(
```

```
        p.html(
```

```
            p.body(
```

```
                p.h1("Thanks!"),
```

```
                p.p("Your items have been ordered.")
```

```
            )
```

```
        )
```

JSON –

To store data in a file

To send/receive data between browser and server

For APIs and web apps

JSON FILE STORAGE & RETRIEVAL: # last page for another json method

```
import os
```

```
import json
```

```
# STEP 1: SET FILENAME
```

```
FILENAME = "data.json" # Change to "orders.json", "players.json", etc.
```

```
# STEP 2: LOAD DATA FROM A JSON FILE
```

```
def load_data():
```

```
    if os.path.exists(FILENAME): # Loads and returns data (usually a list of dictionaries).
```

```
        with open(FILENAME, "r") as f:
```

```
            return json.load(f)
```

```
    return []
```

```
    # Returns an empty list if the file doesn't exist.
```

```
# STEP 3: SAVE DATA TO A JSON FILE
```

```
def save_data(data)
```

```
with open(FILENAME, "w") as f: # Writes (overwrites) the entire list/dict to the file.
    json.dump(data, f, indent=4)
```

STEP 4: ADD NEW DATA (E.G. IN A POST REQUEST)

```
def add_entry(new_entry): # Loads existing data, adds the new entry, then saves it back.
    data = load_data()
    new_order = { # create a new dictionary
        "segments": segment_list,
        "total": total
    }
    data.append(new_order) # add new dictionary to file
    save_data(data)
```

STEP 5: EXTRACT / SEARCH / FILTER FUNCTIONS

Find a single entry by a specific field

```
def find_by_name(data, name):
    for entry in data:
        if entry["name"/"id"/"email"] == name/id/email:
            return entry
    return None
```

Get all values for a specific key (e.g. all names)

```
def get_all_names(data):
    return [entry["name"] for entry in data]
```

Filter values by condition (e.g. score > threshold)

```
def get_high_scores(data, threshold):
    return [entry["score"] for entry in data if entry["score"] > threshold]
```

TO CLEAR A JSON FILE:

```
@app.get("/clear")
```

```
def clear_data():
```

```
    save_data([]) # Overwrite data file with an empty list
```

```
    return str(
        p.html(
            head_html("Data Cleared"),
            p.body(
                navbar_html(),
                p.h1("All players have been removed."),
                p.p(a(href="/")("Back to Home"))
            )
        )
    )
    ...
```

Server-side vs Client-side –

Client-side:

Code runs on the user's device (in the browser).

Faster feedback, but less secure

Good for displaying content, form validation, animations, etc.

Data can be viewed or changed by users

Server-side:

Code runs on the server (before being sent to the user's browser).

More secure and powerful

Used for storing data, processing logic, handling databases, authentication, etc.

Users only see the result, not the code

Functions –

Look for keywords in the question:

"each" → loop (for)

"sum" or "total" → use +=

"find something" → use if, return

"rounded" → use round(number, 2)

"add" → .append()

"average" → divide by len()

"first" → return early inside loop

"highest" or "lowest" → use comparison + temporary variable

"check if" → return True/False

"filter" → return a list based on condition

"update" → modify dictionary or list in-place

"group by" or "count by" → use dictionary to tally

"top" or "rank" → use sorted(..., key=lambda x: x["field"], reverse=True) # reverse=True for highest to lowest

Data Type	Example key	Notes
Strings	key=lambda x: x["name"]	Alphabetical order (A–Z)
Numbers	key=lambda x: x["score"]	Smallest to largest (unless reverse=True)

Dates	key=lambda x: x["date"]	If dates are datetime objects or in sortable format
-------	-------------------------	---

Custom logic	key=lambda x: len(x["text"])	You can rank by anything, even length or calculations
--------------	------------------------------	---

Sort or rank (e.g. top 3 players/aphabetical)

```
def get_top_players(players, n):
    return sorted(players, key=lambda p: p["elo"], reverse=True)[:n] # takes the first n player
```

Calculate total cost

```
def calculate_book_cost(order):
    total = 0
    for book in order:
        total += book["price"] * book["quantity"]
    return round(total, 2)
```

To call values from dictionary

```
def calculate_scarf_cost(scarf_info):
    total_cost = 0
    for segment in scarf_info:
        colour = segment["colour"]
        length = segment["length"]
        cost_per_cm = colour_costs[colour]
        total_cost += cost_per_cm * length
    return round(total_cost, 2)
```

Calculate average

```
def get_average(students):
    total = 0
    for student in students:
        total += student["grade"]
    average = total / len(students)
    return round(average, 1)
```

Add item to list

```
def add_product(cart, id, name, price):
    product = {
        "id": id,
        "name": name,
        "price": price,
        "quantity": 1
    }
    cart.append(product)
```

Find an item

```
def get_student(students, name):
    for student in students:
        if student["name"] == name:
            return student
```

Return highest value

```
def get_largest(transactions):
    largest = transactions[0]
    for t in transactions:
        if t["amount"] > largest["amount"]:
            largest = t
    return largest
```

OR

```
def find_highest(data, key):
    highest_value = float("-inf") # Start with smallest possible value
    top_item = None
    for item in data:
        if item[key] > highest_value:
            highest_value = item[key]
            top_item = item
    return top_item
```

Return lowest value

```
def get_lowest_score(scores):
    lowest = scores[0]
    for score in scores:
        if score < lowest:
            lowest = score
    return lowest
```

OR

```
def find_lowest(data, key):
    lowest_value = float("inf") # Start with largest possible value
    bottom_item = None
    for item in data:
        if item[key] < lowest_value:
            lowest_value = item[key]
            bottom_item = item
    return bottom_item
```

Count how many match condition

```
def count_passes(students):
    count = 0
    for student in students:
        if student["grade"] >= 50:
```

```

    count += 1
    return count

```

Outcome-based (if the question says if ... if ... otherwise ...)

```

if result == "win":
    outcome_factor = 1
elif result == "loss":
    outcome_factor = 0
else: # e.g. "draw" or unknown
    outcome_factor = 0.5

```

Boolean check (return True/False)

```

def is_valid_email(user):
    return "@" in user["email"]

```

Filter items (return subset list)

```

def get_passing_students(students):
    passing = []
    for s in students:
        if s["grade"] >= 50:
            passing.append(s)
    return passing

```

Modify in-place (e.g. update quantity)

```

def update_quantity(catalogue, id, new_qty):
    for item in catalogue:
        if item["id"] == id:
            item["quantity"] = new_qty
    Return

```

Group by category (e.g. count per faculty)

```

def count_by_faculty(players):
    counts = {}
    for p in players:
        faculty = p["faculty"]
        if faculty not in counts:
            counts[faculty] = 0
        counts[faculty] += 1
    return counts

```

To call a function:

```

result = your_function()
print(result)

```

FORMS –

STEP 1: DISPLAY THE FORM (GET request)

```

@app.get("/")
def homepage():
    return str(
        p.html(
            p.head(
                p.title("Beautiful Form"),
                p.link(rel="stylesheet", href="/static/style.css")
            ),
            p.body(
                p.h1("Fill Out the Form!"),
                p.div(class_="form-box")(
                    p.form(action="/", method="post")( # Always include action + method!
                        # Text input
                        p.p("Enter your name:"),
                        p.input(type="text", name="name", placeholder="Your Name",
                            required=True),

                        # Number input
                        p.p OR p.label("Enter your age:"),
                        p.input(type="number", name="age", placeholder="Your Age",
                            required=True),

                        # Dropdown select
                        p.p("Favourite Colour:"),
                        p.select(name="colour")(
                            p.option(value="red")("Red"),
                            p.option(value="blue")("Blue"),
                            p.option(value="green")("Green"),
                            p.option(value="yellow")("Yellow"),
                        ),
                        OR
                        p.option(f) for f in UNSW_FACULTIES # using variables from list/dict

                        # Checkboxes
                        p.p("Choose your hobbies:"),
                        p.label(
                            p.input(type="checkbox", name="hobbies", value="reading"),
                            "Reading"
                        ),
                    )
                )
            )
        )

```

```

p.label(
    p.input(type="checkbox", name="hobbies", value="sports"),
    "Sports"
),
p.label(
    p.input(type="checkbox", name="hobbies", value="music"),
    "Music"
),
p.label(
    p.input(type="checkbox", name="hobbies", value="travel"),
    "Travel"
),

# Submit button
p.br(), p.br(),
p.input(type="submit", value="Submit Form")
)
....

```

TWO BUTTONS IN THE SAME FORM – DIFFERENT ROUTES

```

# Form 1 → Submit to POST /
p.form(action="/", method="post")(
    # form fields...
    p.input(type="submit", value="Calculate")
)

# Form 2 → Submit to POST /order
p.form(action="/order", method="post")(
    p.input(type="submit", value="Order Scarf")
)

```

STEP 2: HANDLE THE FORM SUBMISSION (POST request)

```

@app.post("/")
def submit_form():
    amount = float(request.form["amount"])
    percentage = int(request.form["percentage"])

    # FOR DROPDOWN:
    request.form.get("choice", "") # Safe: returns string or default (" " if not found)

    tip = get_tip(amount, percentage)
    total = get_total(amount, percentage)

    return str(
        p.html(
            p.body(
                p.h1("Results"),
                p.p(f"Subtotal = ${amount:.02f}"),
                p.p(f"Tip of {percentage}% = ${tip:.02f}"),
                p.p(f"Total = ${total:.02f}"),
                p.p(p.a(href="/")("Try again?"))
            )
        )
    )
....

```

HOW TO CALL FUNCTIONS

```

@app.post("/route")
def handle_form():
    # 1. Get data from the form
    value1 = float(request.form["field1"])
    value2 = request.form["field2"]

    REMEMBER: request.form returns strings
    Make sure to wrap in float() to prevent errors

    # 2. Call your function with the data
    result = your_function(value1, value2)

    # 3. Use the result (e.g. return it or redirect)

```

```

return str(
    p.html(
        p.head(
            p.title("Tip Calculator")
        ),
        p.body(
            p.h1("Simple Tip Calculator!"),
            # etc
            p.p(f"Result is: {result}"),
        )
    )
)

```


TO REDIRECT IN WEBPAGES:

```
return redirect("/") # redirect to homepage
```

TO DIRECT TO ANOTHER PAGE:

```
E.g. if action == "Order Scarf":
    return order_page()
```

```
@app.get("/order")
def order_page():
    ...
```

HTML –

A **markup language** – used to describe a collection of content. Has information about what should be displayed on the webpage.
HTML is used to create and show a web page as well as different elements and features of the web page. For example, HTML is used to create a form on a web page that allows a users to submit name, email, and phone number.

CSS –

CSS is a **styling language** – used to design and describe the visual appearance of a HTML webpage. For example, CSS will be utilised to design the form made with HTML such that it improves aesthetics and usability. The font type and size can be changed along with the colours of the page.

```
/* == GLOBAL RESET AND FONT == */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: "Segoe UI", sans-serif;
    background-color: linen;
    color: black;
    line-height: 1.6;
    padding: 20px;
}

/* == HEADER AND NAVIGATION == */
header {
    background-color: maroon;
    color: white;
    padding: 20px;
    text-align: center;
    font-size: 2rem;
    border-radius: 8px;
    margin-bottom: 30px;
}

nav {
    margin-bottom: 20px;
    text-align: center;
}

nav a {
    text-decoration: none;
    color: maroon;
    margin: 0 15px;
    font-weight: bold;
}

nav a:hover {
    text-decoration: underline;
}

/* == MAIN CONTENT AREA == */
main {
    max-width: 800px;
    margin: 0 auto;
    background-color: white;
    padding: 30px;
    border-radius: 12px;
    box-shadow: 0 0 10px gainsboro;
}

/* == HEADINGS AND TEXT == */
h1, h2, h3 {
    color: maroon;
    margin-bottom: 10px;
    text-align: center;
}

/* == FORM STYLING == */
form {
    margin-top: 20px;
    display: flex;
    flex-direction: column;
    gap: 10px;
}
```

OR

```
.form-box {
    max-width: 500px;
    margin: 0 auto;
    padding: 20px;
    background-color: white;
    border-radius: 12px;
    box-shadow: 0 0 10px #ccc;
    text-align: center;
}

input[type="text"],
input[type="email"],
input[type="password"],
textarea,
select {
    padding: 10px;
    border: 1px solid darkgray;
    border-radius: 6px;
    font-size: 1rem;
}

input[type="submit"],
button {
    background-color: maroon;
    color: white;
    border: none;
    padding: 10px 16px;
    font-size: 1rem;
    border-radius: 6px;
    cursor: pointer;
    transition: background-color
0.2s;
}

input[type="submit"]:hover,
^ MAKE SURE TO ADD save_data(data) to ALL app.post(/add...)
button:hover {
    background-color: firebrick;
}

/* == TABLES == */
table {
    width: 100%;
    border-collapse: collapse;
    margin-top: 20px;
}

table, th, td {
    border: 1px solid silver;
}

th, td {
    padding: 10px;
    text-align: left;
    background-color: floralwhite;
}
```

```
def get_data():
    """
    Load the data from 'tournament.json'
    """
    T000: T7
    try:
        return json.load(open("tournament.json"))
    except Exception:
        return []

def save_data(data):
    """
    Save the data to 'tournament.json'
    """
    T000: T7
    json.dump(data, open("tournament.json", "w"))
```

Factor	Use Cookies	Use File / Database
Data size	Small (≤ 4KB)	Large or complex
Where data is stored	On user's browser (client-side)	On server (backend)
Data purpose	Remember user preferences, session ID, etc.	Store critical info, logs, rankings, form submissions
Security importance	Low – data can be read/edited by user	High – private/sensitive data needs secure storage
Persistence across users/sessions	No – only for current user	Yes – share and persist across users and sessions
Persistence across devices	No – tied to one browser	Yes – accessible from multiple devices

Error Type	Cause	Error Message	Fix	Why the Fix Works
KeyError	Trying to access a dictionary key that doesn't exist	KeyError: 'bob'	Use .get() users.get("bob") Add username to dictionary	.get() returns None if the key doesn't exist, preventing a crash.
IndexError	Accessing an index in a list/tuple/string that's out of range	IndexError: list index out of range	Check len() first if len(my_list) > 5: print(my_list[5])	Checks if the index exists before accessing it.
TypeError	Performing an invalid operation on a type (e.g. adding str + int)	TypeError: can only concatenate str ('str') to str	Convert types properly: int(...), str(...), etc. int("20") + 5	Converts string to integer so the types match.
ValueError	Passing a wrong value to a function (e.g. int("abc"))	ValueError: invalid literal for int()	Validate input before converting if string.isdigit(): int(string)	Checks if the string contains only digits before converting.
AttributeError	Accessing an attribute/method that doesn't exist on the object	AttributeError: 'int' object has no attribute 'append'	Check object type before calling method my_list = [5]my_list.append(3)	Only lists have .append(), not integers.
NameError	Using a variable that hasn't been defined	NameError: name 'total' is not defined	Make sure all variables are defined beforehand total = 0	Variable must be defined before it is used.