Importance ■■■■■ most asked ■■■□□ often ■□□□□ rarely

Stack program (3 marks)

A **stack** is LIFO: Last In, First Out. In Python it is a list: **push** = `append()` adds to the top (the end), **pop** = `pop()` removes from the top.

The usual question: (I) **push** the items that match a condition from a given list or dictionary, (II) **pop and display** them until the stack is empty, then print a message.

(I) Push the matching records · (II) pop until empty

```
L = [("Laptop", 90000), ("Mobile", 30000),
     ("Pen", 50), ("TV", 60000)]
Product = []
def Pushelement():
    for name, price in L:
        if price > 50000:
            Product.append((name, price))
def Popelement():
    while Product:
        print(Product.pop())
    print("Stack Empty")
Pushelement()
Popelement()
```

```
('TV', 60000)
('Laptop', 90000)
Stack Empty
```

IF THE QUESTION PUSHES…	USE	ASKED
records (tuples, lists) that match	`for name, price in L: · if price > 50000:`	■■■□□
numbers that match	`for n in N: · if n % 2 == 0:`	■■■□□
dictionary keys whose value matches	`for k in D: · if D[k] > 100: · push k`	■■□□□

One pop with underflow, peek, display

```
def pop(stack):
    if stack == []:
        print("Underflow")
        return None
    return stack.pop()
def peek(stack):
    if stack == []:
        return None
    return stack[-1]
def display(stack):
    for i in range(len(stack) - 1, -1, -1):
        print(stack[i])
s = [10, 20, 30]
print(pop(s), peek(s))
display(s)
print(pop([]))
```

```
30 20
20
10
Underflow
None
```

! `pop()` on an empty list raises **IndexError**, so check `if stack == []` first. The top is the **last** element, `stack[-1]`; display it first. Popped items come out in the **reverse** of the order they went in.

Global and local scope

A variable made **inside** a function is **local**: it exists only while the function runs. A variable made **outside** every function is **global**. A function can read a global, but assigning to it makes a new local, unless you declare it `global`.

```
x = 10
def f():
    global x
    x = 20
f()
print(x)
```

```
20
```

```
y = 8
def update():
    y = y + 2
    print(y)
update()
```

```
UnboundLocalError: cannot access local variable 'y'
where it is not associated with a value
```

! `y = y + 2` inside a function makes y local, so reading it first fails: **UnboundLocalError**, not 10. Without the `global x` line, the first program would print **10**.

Function output questions

Flow of execution: Python skips a `def` until the function is **called**; then it runs the body, and `return` sends a value back to where it was called.

```
def changer(p, q=10):
    p = p / q
    q = p % q
    print(p, '#', q)
    return p
a = 200
b = 20
a = changer(a, b)
print(a, '$', b)
a = changer(a)
print(a, '$', b)
```

```
10.0 # 10.0
10.0 $ 20
1.0 # 1.0
1.0 $ 20
```

```
def change(L, n):
    L.append(n)
    n = n + 1
nums = [1, 2]
x = 5
change(nums, x)
print(nums, x)
```

```
[1, 2, 5] 5
```

! / always gives a **float**: write 10.0, not 10. A list passed in and changed with `append` changes for the caller too; a number reassigned inside does **not**. `a = changer(a)` replaces a with the returned value.

Returning values

```
def min_max(L):
    return min(L), max(L)
r = min_max([4, 9, 1])
print(r)
lo, hi = min_max([4, 9, 1])
print(lo, hi)
def no_return():
    x = 5
print(no_return())
```

```
(1, 9)
1 9
None
```

Several values come back as a **tuple**; unpack them into names. A function with no `return` returns **None**.

Write a function (2 marks)

Usually: accept a list or dictionary as a **parameter**, check a condition, change it or print a message.

```
def updatescore(scores, name, new):
    if name in scores:
        scores[name] = new
    else:
        print(name, "not found")
scores = {'Asha': 40, 'Ravi': 35}
updatescore(scores, 'Ravi', 50)
updatescore(scores, 'Mohan', 20)
print(scores)
```

```
Mohan not found
{'Asha': 40, 'Ravi': 50}
```

try, except, finally

An **exception** is an error while the program runs. `try` holds the risky code; `except` handles the error; `finally` **always** runs, error or not.

```
def divide(a, b):
    try:
        print(a / b)
    except ZeroDivisionError:
        print("Cannot divide by 0")
    finally:
        print("Done")
divide(10, 4)
divide(10, 0)
```

```
2.5
Done
Cannot divide by 0
Done
```

```
try:
    x = int("abc")
except ValueError:
    print("Value error!")
except Exception:
    print("General error!")
```

```
Value error!
```

! Only the **first** matching `except` runs, so put specific ones before `Exception`. A `try` needs at least one `except` **or** a `finally`; `try` + `finally` alone is valid. A **syntax error** (missing colon) stops the program before it runs and can't be caught this way; ZeroDivisionError is not a syntax error.

Built-in exceptions

CODE	RAISES
`10 / 0`	ZeroDivisionError
`int('abc')`	ValueError: right type, wrong value
`'a' + 5`	TypeError: wrong type for the operation
`[1, 2][5]`	IndexError
`{'a': 1}['b']`	KeyError
`undefined_name`	NameError
`import nomodule`	ModuleNotFoundError, a kind of ImportError
`pickle.load()` at the end of a file	EOFError

Arguments, parameters, types of functions

TERM	MEANS
Parameter	the name in the `def` line: `def changer(p, q=10)`
Argument	the value in the call: `changer(200, 20)`
Positional	matched in order: 200 → p, 20 → q
Default	`q=10` is used when no value is given
Named	matched by name, any order: `changer(q=5, p=50)`

```
def f(a=1, b):
    return a + b
```

```
SyntaxError: parameter without a default follows
parameter with a default
```

! Default parameters must come **after** the others: `def f(b, a=1)` is right.

Types of functions: **built-in** (`len()`, `print()`), defined in a **module** (`math.sqrt()`, after `import math`), and **user-defined** (made with `def`).

What's the output?

Cover the right column and answer; every result was run.

CODE	OUTPUT

`def f(a, b=2):`
`  return a * b`
`print(f(3), f(3, 4))`
	6 12

`def g(n):`
`  n += 1`
`  return n`
`x = 5`
`print(g(x), x)`
	6 5

`st = []`
`for ch in 'ABC':`
`  st.append(ch)`
`print(st.pop() + st.pop())`
	CB

Find the error

WRONG	RIGHT
`Def add(a, b):`	`def add(a, b):`
`def f(a, b)`	`def f(a, b):`
`except ValueError`	`except ValueError:`
`stack.push(5)`	`stack.append(5)`
`stack.pop()` on an empty stack	check `if stack != []` first
`stack.pop(0)` for the top	`stack.pop()`