

Lecture 6

Loops – repeating instructions with `while` and `for`

Lecture 6 - Loops

- Learning objectives
 - understand when repetition is needed in a program
 - write and trace `while` and `for` loops
 - use counters, accumulators, sentinels and flags
 - recognize infinite loops and off-by-one errors
 - use `range()` and iterate over strings
 - combine loops with conditions and nested blocks

Lecture 6 - Loops

- Why do we need loops?
 - Programs often repeat the same operations with different values
 - We may not know how many repetitions are needed
 - Writing the same statement many times is error-prone and does not scale.
 - A loop describes the repetition once and lets Python repeat it.

Without a loop

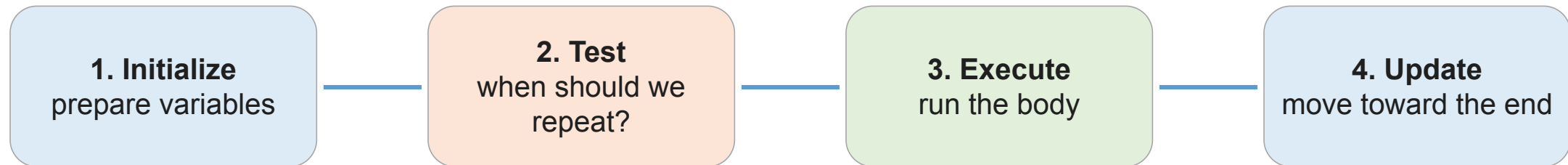
```
print(1)
print(2)
print(3)
print(4)
print(5)
```

With a loop

```
for i in range(1, 6):
    print(i)
```

Lecture 6 - Loops

- A well-designed loop usually has four clearly identifiable parts.



The update must eventually make the loop condition False (Stop).

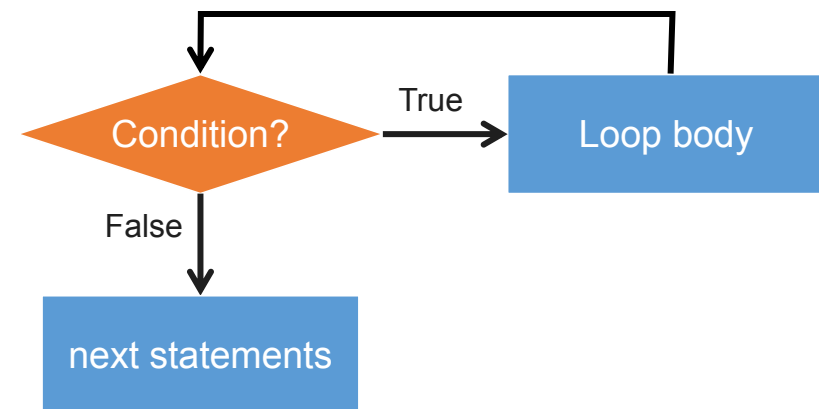
The *while* loop

Repeat while a condition remains True

Lecture 6 - The while loop

- Syntax
 - ‘:’ and **indentation** to separate header and body
 - The condition is tested **before** every iteration (first time before entering the loop, as an **if** statement)
 - The loop may execute zero times.
 - The body executes only while the condition is **True**.

```
Python
while condition:
    statement_1
    statement_2
next_statement
```



Lecture 6 - The while loop

- Example problem: investment growth (see example in lecture 1)
 - An account earns a fixed annual interest rate.
 - *How many years are required for the balance to reach a target?*
 - The number of iterations is not known in advance → while is a natural choice.

Year	Balance
0	10000
1	$10000 * 1.05 = 10500$
2	$10500 * 1.05 = 11025$
...	...
14	$18856 * 1.05 = 19799$
15	$19799 * 1.05 = 20789 > 20000$

Unknown number of years → condition-controlled loop

Lecture 6 - The while loop

- From pseudocode to Python

1. Set value of year to 0
2. Set value of balance to 10000
3. **While** balance < 20000 do:
 1. add 1 to the value of year
 2. Multiply balance for 1.05
4. Return the value of year

● ● ● double_investment.py

```
RATE = 5.0
INITIAL_BALANCE = 1000.0
TARGET = 2 * INITIAL_BALANCE

year = 0
balance = INITIAL_BALANCE

while balance < TARGET:
    year += 1
    balance += balance * RATE / 100

print(f"Doubled after {year} years")
```

Lecture 6 - The while loop

- From pseudocode to Python

1. Set value of year to 0
2. Set value of balance to 10000
3. **While** balance < 20000 do:
 1. add 1 to the value of year
 2. Multiply balance for 1.05
4. Return the value of year

In Python, `x += y` is
the short form for:

`x = x + y`

```
double_investment.py

RATE = 5.0
INITIAL_BALANCE = 1000.0
TARGET = 2 * INITIAL_BALANCE

year = 0
balance = INITIAL_BALANCE

while balance < TARGET:
    year += 1
    balance += balance * RATE / 100

print(f"Doubled after {year} years")
```

Lecture 6 - Hand-tracing a loop

- Hint: *Hand-tracing* is a useful method in both phases:
 - **design** of the algorithm
 - **test** of the program
- Hand-tracing means executing the program manually, one iteration at a time.
- Track the variables that determine the loop state and update them row by row.

Iteration	year	balance	balance < 2000
start	0	1000.00	True
1	1	1050.00	True
2	2	1102.50	True
3	3	1157.63	True
...	...	...	...

Tracing helps detect wrong updates and wrong conditions.

Lecture 6 - while loop patterns

Counter-controlled loop

- The number of repetitions is **known**
- A counter is initialized, tested and updated
- Possible with `while`, although `for` is often clearer (see next slides).

```
counter = 1
while counter <= 5:
    print(counter)
    counter += 1
```

Event/condition-controlled loop

- The number of repetitions is **not known** in advance
- The loop stops when a condition becomes `False`
- Typical for convergence, user input and simulations.

```
while balance < target:
    year += 1
    balance *= 1 + rate / 100
```

Lecture 6 - while loop patterns

Mixed Counter- and Condition-controlled loop

- Example: read a PIN code from user. The program ends after 5 tentatives, OR if the PIN is correct.

```
PIN = 1234
count = 1
correct = False

while (count <= 5) and not correct:
    read = int(input('PIN: '))
    if read == PIN:
        correct = True
        print('correct')
    else:
        print(f'Tentative {count+1}')
    count = count + 1
```

Lecture 6 - Choosing the test condition

- The body executes only when the condition is True.
- Ask: “What must still be true for another iteration to be needed?”
- For a balance that grows toward TARGET, continue while `balance < TARGET`.

Write the condition in words before writing it in Python

• • • Correct

```
while balance < TARGET:  
    balance *= 1.05
```

• • • Wrong direction

```
while balance >= TARGET:  
    balance *= 1.05
```

Lecture 6 - Common error: infinite loops

- A while loop runs until its condition becomes False
- If the tested variable never changes in the required direction, the loop may **never stop**
- Check that every iteration makes progress toward termination

● ● ● Infinite loop

```
balance = 1000
TARGET = 2000

while balance < TARGET:
    year += 1
    # balance is never updated!
```

No progress → condition stays True forever

Lecture 6 - Common error: infinite loops

- A `while` loop runs until its condition becomes `False`
- If the tested variable never changes in the required direction, the loop may **never stop**
- Check that every iteration makes progress toward termination

```
• • • Infinite loop

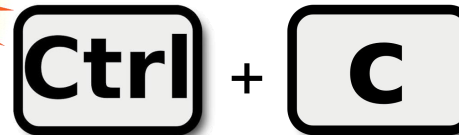
balance = 1000
TARGET = 2000

while balance < TARGET:
    year += 1
    # balance is never updated!
```

An infinite loop **do not raise errors** in Python: the program keeps executing the loop until it is externally interrupted (e.g. CTRL+C) or another condition terminates it (e.g. integer overflow, or it eventually exhausts some resource, such as memory)



No progress → condition stays `True` forever



Lecture 6 - Common error: off-by-one

- An off-by-one error executes the body one time too many or one time too few.
- Typical causes:
 - starting a counter at the wrong value
 - using `<` instead of `<=`, or vice versa
 - forgetting that `range()` excludes its stop value (we will see this function in the following)

Print 1, 2, 3, 4, 5

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Before coding, decide exactly which values must be included.

Lecture 6 - Early interruption: break and continue

- `break` ends the *nearest loop* (the closest taht surrounds the break) immediately
- `continue` skips the rest of the current iteration and starts the next one
- Use them when they make the control flow clearer, not to hide a poorly designed condition.

```
for attempt in range(1, 4):  
    pin = input("PIN: ")  
    if pin == "4827":  
        print("Access granted")  
        break  
  
    if pin == "":  
        continue  
  
    print("Wrong PIN")
```

Sentinels and loop patterns

Processing a sequence when its length is not known in advance

Lecture 6 - Sentinel-controlled input

- A sentinel is a special value that marks the end of the data
- The sentinel is not part of the data being processed; it is part of the strategy we apply (a convention we define)
- Useful when the number of inputs is not known beforehand.

Choose a sentinel that cannot be confused with valid data.

● ● ● Sentinel value: -1

```
total = 0.0  
count = 0
```

```
salary = float(input("Salary (-1 to finish): "))  
while salary != -1:  
    total += salary  
    count += 1  
    salary = float(input("Salary (-1 to finish): "))
```

Lecture 6 - Accumulators and counters

- Many loop algorithms use a small set of recurring variables:
 - **counter** → how many values have been processed
 - **accumulator** → running sum or total
 - **flag** → whether an event has occurred
 - **current best** → minimum or maximum seen so far

count = 0
count += 1

total = 0
total += value

found = False
found = True

largest = first
if value > largest:
largest = value

Lecture 6 - Sum and average

- To compute an **average**, keep both the running sum and the number of values.
- Always handle the case `count == 0` before dividing.

```
total = 0.0
count = 0

value = float(input("Value (-1 to finish): "))
while value != -1:
    total += value
    count += 1
    value = float(input("Value (-1 to finish): "))

if count > 0:
    print(f"Average: {total / count:.2f}")
```

Lecture 6 - Tracking a minimum or maximum

- The first valid value can initialize the current minimum/maximum
- Each following value is compared with the best value seen so far
- This avoids inventing a “magic” initial number.

```
first = float(input("First value: "))
largest = first

value = float(input("Next value (-1 to finish): "))
while value != -1:
    if value < largest:
        largest = value
    value = float(input("Next value (-1 to finish): "))

print("Largest: ", largest)
```

The *for* loop

Iterating over a sequence of values

Lecture 6 - The for ... in loop

- A for loop visits each value produced by an **iterable** object
- Strings are already iterable: the loop can process one character at a time
- Use for when the set or number of iterations is naturally known
- NOTE: similar to a counter-controlled while!

```
word = "Python"  
  
for ch in word:  
    print(ch)  
  
print("Done")
```



ch takes one character at each iteration

Lecture 6 - range()

- `range(stop)` produces (iterable) integers from 0 up to `stop - 1`
- The stop value is **excluded**
- This matches Python indexing, which also starts at 0
- NOTE: the value returned from `range()` is of type 'range' (not list or tuples that we will see), check it with `type()`!

```
Python
for i in range(5):
    print(i)
```



0, 1, 2, 3, 4 — not 5

Lecture 6 - range(start, stop, step)

- range() can specify the **first value**, the **excluded stop value** and the **step**.

● ● ● start, stop

```
for i in range(2, 7):  
    print(i)
```

● ● ● positive step

```
for i in range(0, 11, 2):  
    print(i)
```

● ● ● negative step

```
for i in range(5, 0, -1):  
    print(i)
```

Think in terms of the values you want, then choose start, stop and step.

documentation about the range type: <https://docs.python.org/3/library/stdtypes.html#typeseq-range>

Lecture 6 - while or for?

while

- Use when repetition depends on a condition;
- The number of iterations may be unknown;
- You are responsible for updating the loop state.

```
found = False
i = 0
while not found and i <= 5:
    # some find operation...
    i += 1
```

for

- Use when iterating over known values or a range;
- The loop variable changes automatically;
- Usually clearer for counter-controlled repetition.

```
for i in range(1, 6):
    print(i)
```

Lecture 6 - Index and value with enumerate()

- Sometimes we need both the position and the value
- `enumerate()` generates pairs: index and current element
- This avoids manually maintaining an index counter.
- NOTE: By default, `enumerate()` starts counting from 0. A different starting value can be specified using the `start` parameter:
 - `enumerate(word, start=1)`

```
word = "loop"  
  
for index, ch in enumerate(word):  
    print(index, ch)
```

index	ch
0	l
1	o
2	o
3	p

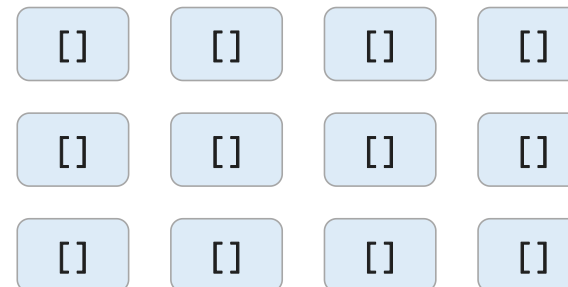
Nested loops and applications

A loop can contain another loop

Lecture 6 - Nested loops

- The inner loop completes all of its iterations for each iteration of the outer loop.
- Each nested block is indented one additional level.
- Typical uses: tables, grids, combinations and repeated datasets, nested searches
 - example: search for a word among multiple files

```
for row in range(3):  
    for column in range(4):  
        print("[ ]", end=" ")  
    print()
```



**3 rows
× 4 columns
= 12 inner
iterations**

Lecture 6 - Searching and Matching in Strings

`in` as a membership operator

- Operator `in` is also used to check whether a value is contained in a sequence or collection.
- For strings, it can be used to verify if a **substring** is contained in another string
- ***Membership test expression*** (that is a kind of *boolean expression*): the `in` operator checks whether a substring is contained in a string and **returns a Boolean value**.

```
"cat" in "concatenate"    # True
"dog" in "concatenate"    # False
```

Lecture 6 - Searching and Matching in Strings

Some useful method for text search

- `.startswith()` e `.endswith()`: check if the initial/final part of a string matches the argument
- `.count()`: counts the number of occurrences in a string of the pattern given as an argument.

```
my_string = 'hello_there.txt'

text.startswith("hello") # True
text.endswith(".txt")    # True
text.count('he')         # 2
```

Lecture 6 - String processing with loops

- A for loop is ideal when every character must be examined
- Combine the loop with if/elif conditions to classify characters
- Counters can summarize the result.

```
text = input("Text: ")
vowels = 0
digits = 0

for ch in text:
    if ch.lower() in "aeiou":
        vowels += 1
    elif ch.isdigit():
        digits += 1

print("Vowels:", vowels)
print("Digits:", digits)
```

Lecture 6 - Loops in simulations

- A simulation repeats a random experiment many times
- The loop controls how many trials are performed
- Counters estimate frequencies and probabilities.
- Example: Monte Carlo method

More repetitions usually produce a more stable estimate.

```
import random

rolls = 1000
sixes = 0

for _ in range(rolls):
    die = random.randint(1, 6)
    if die == 6:
        sixes += 1

print(f"Frequency: {sixes / rolls:.3f}")
```

Lecture 6 - Loops in simulations

- A simulation repeats a random experiment many times
- The loop controls how many trials are performed
- Counters estimate frequencies and probabilities.
- Example: Monte Carlo method

More repetitions usually produce a more stable estimate.

random module (standard) provides function for random number generation, e.g.:
`randint(a, b)` → random integers in a range `[a, b]`

```
import random

rolls = 1000
sixes = 0

for _ in range(rolls):
    die = random.randint(1, 6)
    if die == 6:
        sixes += 1

print(f"Frequency: {sixes / rolls:.3f}")
```

Lecture 6 - A loop-design checklist

- Before writing the code, answer these questions:
 - What exactly is repeated?
 - Do I know the number of iterations? → for may be appropriate
 - If not, what condition should keep the loop running? → while
 - Which variables must be initialized before the loop?
 - How does each iteration update the program state?
 - What guarantees that the loop eventually ends?
 - Which boundary values should I test?

Lecture 6 - Common loop patterns

Counter
count += 1

How many?

Accumulator
total += value

What total?

Sentinel
while value != END

When does input stop?

Search / flag
found = True

Did it happen?

Min / max
if value > best: ...

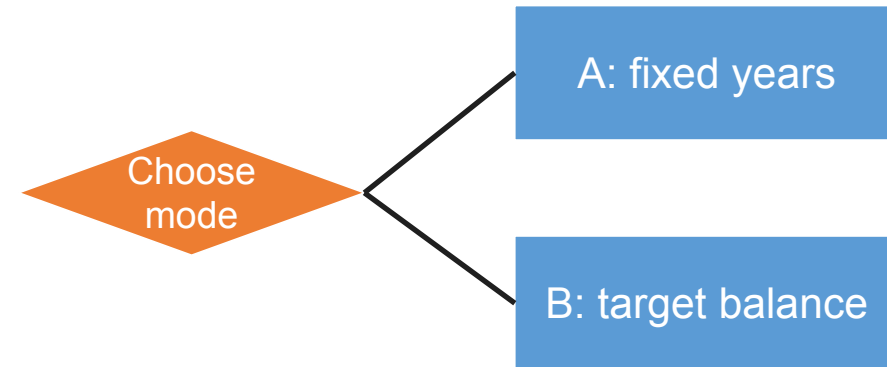
What is the best seen?

Nested loops
for ...:
for ...:

Rows × columns / combinations

Lecture 6 - Guided activity

- Investment Growth Explorer
 - Ask for an **initial balance** and an **annual interest rate**.
 - **Mode A**: compute the balance after a chosen number of years → for loop
 - **Mode B**: compute how many years are required to reach a target → while loop
 - Print a small year-by-year report with formatted balances.



Same formula, different stopping rule:
 $\text{balance} += \text{balance} * \text{rate} / 100$

Lecture 6 - Practice notebooks

- Three notebooks accompany this lecture:
 - **Guided:** Investment Growth Explorer
 - **Laboratory:** sentinel statistics + string character profiler
 - **Independent practice:** digit analysis, PIN attempts, dice simulation, nested power table
- A slide deck presents the problem and some hints for the implementation

Lecture 6 - Recap

- `while` repeats while a Boolean condition is True
- `for` iterates over a sequence of values; `range()` is ideal for integer-controlled repetition
- Counters, accumulators, flags, sentinels and min/max tracking are reusable loop patterns
- Always reason about initialization, condition, body and update
- Trace boundary cases carefully: **infinite loops** and **off-by-one errors** are common
- Loops can be **nested** and combined with conditions to build non-trivial programs.