

```
$ python compare.py  
What's x? 1    What's y? 2  
x is less than y
```

## Lecture 2 · Conditionals and Loops

RUHR-UNIVERSITÄT BOCHUM

# INTRODUCTION TO PYTHON

Prepcourse Computer Science | Prof. Dr. Nils Jansen | Faculty of Computer Science

# Acknowledgements

## Source material

- This lecture is based on CS50's Introduction to Programming with Python (CS50P), Harvard University — David J. Malan and the CS50 team.
- Structure, examples, and terminology follow Lecture 1: Conditionals and Lecture 2: Loops.
- CS50 course materials are licensed under CC BY-NC-SA 4.0. Course site: <https://cs50.harvard.edu/python/>

## Use of AI

- Claude (Anthropic) was used for initial drafting of these slides.
- All content was reviewed, modified, and verified by Nils Jansen, who is responsible for the final version.

# Agenda

Conditions that choose a branch, loops that repeat work

**1**

## Recap

types, variables, functions

**2**

## Comparisons

<, ==, != and friends

**3**

## if / elif / else

pick one branch

**4**

## and / or

combine two conditions

**5**

## match

one value, many cases

**6**

## while

repeat until done

**7**

## for and range

repeat a set number

**8**

## Lists and dicts

loop over your data

## **9 Challenges** the Sorting Hat and Mario's pyramid

# Lecture 1 in one short program

## recap.py

```
name = input("Name? ").strip().title()
age = int(input("Age? "))
print(f"hello, {name}")
print(f"Next year you are {age + 1}")
```

## terminal

```
$ python recap.py
Name? ada      Age? 30
hello, Ada
Next year you are 31
```

**Functions.** `print`, `input` and `int()` are called by name, arguments in brackets.

**Variables store values.**

**Methods clean up.** `.strip()` and `.title()` are called on the value with a dot.

**f-strings build text.** `{}` drops a value into the sentence.

**Then: `def` and `return`** let you name your own actions.

# Every value has a type: int, float, str, bool

types.py

```
x = 7          # int, a whole number
y = 2.5        # float, a decimal
name = "Ada"   # str, text in quotes
ok = True      # bool, True or False
```

check it with type()

```
print(type(7))      # <class 'int'>
print(type("7"))    # <class 'str'>
```

**int counts things.** Whole numbers, no decimal point: 0, 7, -3.

**float measures things.** It has a decimal point and is stored approximately.

**str is text.** Quotes make it text, so "7" is not the number 7.

**bool is a truth value.** True or False, the result of every comparison.

# A variable names a value; the type is automatically set

names.py

```
x = 7          # a name for 7
x = x + 1      # 8, still an int
x = "seven"    # now a str
```

converting on purpose

```
int("7") + 1   # 8
str(7) + "!"   # "7!"
int(2.9)       # 2, it cuts
"7" + 1        # TypeError
```

**A variable is a name.** = stores the value on the right under that name.

**Python picks the type.** You never declare it; the value decides.

**input always gives str.** Convert with `int()` or `float()` before you calculate.

**`int()` cuts, `round()` rounds.**  
`int(2.9)` is 2, `round(2.9)` is 3.

# Learning goals

By the end of the session you can

- ✓ **Write a condition** and say whether it is True or False.
- ✓ **Choose a branch** with `if`, `elif` and `else`.
- ✓ **Combine conditions with** `and`, `or` and `0 <= x <= 10`.
- ✓ **Repeat work** with `while` and `for`, and stop it with `break`.
- ✓ **Loop over data** held in a list or a dict.
- ✓ **Return a bool** from your own function, like `is_even`.

# Part 1

# Conditionals



1 · Conditionals



2 · Loops

# if runs a block when its condition is True

## compare.py

```
x = int(input("What's x? "))
y = int(input("What's y? "))

if x < y:
    print("x is less than y")
```

## the six comparisons

```
x < y      x <= y      x == y
x > y      x >= y      x != y
```

### **A condition is an expression.**

Python runs the block only when it evaluates to True.

**Colon, then indent.** The indented lines are the block; the indent is the syntax.

**== compares, = assigns.** Don't mix them up.

**One if, one block.** If the condition is false, Python skips straight past it.

# elif becomes active if the previous condition failed

## three conditions, three branches

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
elif x == y:
    print("x is equal to y")
```

## one condition fewer

```
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

**elif is checked in order.** It is checked only when every condition above it was False.

**The chain stops at the first hit.** Separate ifs would all be activated.

**else has no condition.** It catches whatever is left.

**If x is not smaller and not bigger,** it is equal. Else catches that case.

# or needs one side true; and needs both

**or — either one is enough**

```
if x < y or x > y:
    print("x is not equal to y")
else:
    print("x is equal to y")
```

**and — both must hold**

```
if score >= 90 and score <= 100:
    print("Grade: A")
```

**or is true** as soon as one side is true.

**and is true** only when both sides are.

**Python stops early.** Once the result is determined, the rest is not evaluated.

**Fun tip: Say it out loud.** If the sentence is clear, the condition usually is too.

# 90 <= score <= 100 means what it looks like

grade.py

```
score = int(input("Score: "))

if 90 <= score <= 100:
    print("Grade: A")
elif 80 <= score < 90:
    print("Grade: B")
elif 70 <= score < 80:
    print("Grade: C")
else:
    print("Grade: F")
```

**Two comparisons in one.** Python allows the maths notation, and it reads the same.

**Order matters.** The first matching branch wins, so go from the top down.

**Upper bounds are optional.** Anything higher was caught above.

**else is the safety net.** Every other score lands there.

# A function can return True or False

parity.py

```
def main():
    x = int(input("What's x? "))
    if is_even(x):
        print("Even")
    else:
        print("Odd")

def is_even(n):
    return n % 2 == 0

main()
```

**A comparison is a value.** `n % 2 == 0` is True or False, the two values of type bool.

**return hands it back**, so the function reports a True or False result.

**No if needed inside.** Returning the comparison is enough.

**Name it as a condition.** `if is_even(x)` reads as a condition.

# match compares one value against many cases

house.py

```
name = input("What's your name? ")

match name:
    case "Harry" | "Hermione" | "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
```

**One value, several cases.** Python takes the first case that matches.

**| means or.** Three names, one house, one line.

**case \_ is the catch-all,** the else of a match.

**Use it for one variable** against many values; otherwise stay with `elif`.

# Your turn: two short programs

## task 1 – positive, negative or zero

```
# Ask for a whole number.  
# Print Positive, Negative or Zero.  
# Hint: if / elif / else and > <
```

## task 2 — pass or fail

```
# Ask for a score from 0 to 100.  
# 60 and up passes, below fails.  
# Hint: if / else and >=
```

**Write the comments first**, then fill in one line at a time.

**Run it after every line.** A small error is easier to find than five.

**Try the edge cases:** 0, 59, 60, 100, and a negative number.

**Stuck?** Say the condition out loud, then write exactly that.

# Solutions

## task 1 – sign.py

```
n = int(input("Number? "))
if n > 0:
    print("Positive")
elif n < 0:
    print("Negative")
else:
    print("Zero")
```

## task 2 – grade.py

```
score = int(input("Score? "))
if score >= 60:
    print("Pass")
else:
    print("Fail")
```

**Order matters.** The first condition that is True wins; **else** catches the rest.

**int() first.** **input** returns a **str**, and text cannot be compared with a number.

**Exactly one branch runs.** **elif** is checked only when the conditions above it were **False**.

**Test the edges.** 0 and exactly 60 are the interesting inputs.

# Part 2

# Loops

---

1 · Conditionals

---

2 · Loops

# while repeats while its condition is True

cat.py — meow three times

```
i = 3
while i != 0:
    print("meow")
    i -= 1
meow
meow
meow
```

the loop that never ends

```
i = 3
while i != 0:
    print("meow")
# i never changes – Ctrl-C to stop
```

**The condition comes first.** It is evaluated before every pass, including the first.

**Something must change** inside the loop, or the condition never becomes False.

**i -= 1** is short for **i = i - 1**.

**Counting up reads easier:** **i = 0** and **while i < 3**.

# for takes each value in turn

## a list of values

```
for i in [0, 1, 2]:  
    print("meow")  
meow  
meow  
meow
```

## the same thing with range

```
for i in range(3):  
    print("meow")  
meow  
meow  
meow
```

**One pass per value.** i takes each value in turn, then the loop ends.

**range(3) is 0, 1, 2.** Three values, counted from zero.

**Nothing to initialise,** nothing to update, no way to loop forever.

**Square brackets make a list;** range writes one for you.

# Use `_` when you never read the counter

the counter is not used

```
for _ in range(3):  
    print("meow")
```

or let the string repeat itself

```
print("meow\n" * 3, end="")  
# three lines of meow
```

`_` is a **normal name** that tells the reader: I will not use this value.

**The body never mentions it**, so nothing is lost by hiding it.

`"meow\n" * 3` repeats the string itself, newline included.

`end=""` stops print from adding one more newline.

# break leaves; continue skips one pass

repeat until the input is valid

```
while True:
    n = int(input("What's n? "))
    if n > 0:
        break
```

skip one pass, keep looping

```
for i in range(5):
    if i == 2:
        continue
    print(i)          # 0 1 3 4
```

**while True** loops on purpose forever; break is the way out.

**break ends the loop.** continue ends only the current pass.

**This is how you validate input:**  
read, check, break when the value is valid.

**Inside a function,** return leaves the loop and the function at once.

# First, somewhere to keep several values

a **list** – values in order

```
names = ["Ada", "Alan"]
names[0]          # "Ada"
len(names)        # 2
```

a **dict** – key and value

```
ages = {"Ada": 36}
ages["Ada"]      # 36
```

**Why now:** a loop needs something to repeat over.

**Square brackets make a list.** Items keep their order and are reached by position, starting at 0.

**Curly braces make a dict.** Items are reached by key, not position.

**Other languages say array.** A Python **list** can grow and mix types.

**Not today's topic.** Here we only loop over lists.

# A list holds several values in order

## students.py

```
students = ["Hermione", "Harry", "Ron"]
for student in students:
    print(student)
Hermione
Harry
Ron
```

## when the position matters

```
for i in range(len(students)):
    print(i + 1, students[i])
1 Hermione
2 Harry
3 Ron
```

**Square brackets, commas.** That is a list, and the order stays as you wrote it.

**The loop gives you values,** one per pass, no counter in sight.

**len(students) is 3;** the positions are 0, 1 and 2.

**Only use the index form** when you actually need the number.

# A dict pairs a key with a value

houses.py

```
students = {  
    "Hermione": "Gryffindor",  
    "Harry": "Gryffindor",  
    "Draco": "Slytherin"  
}  
  
for name in students:  
    print(name, students[name])
```

output

```
Hermione Gryffindor  
Harry Gryffindor  
Draco Slytherin
```

**Curly braces**, and every item is key: value.

**Looping gives you the keys**; index the dict to get the value.

`students["Harry"]` looks one up. A key that is not there is an error.

**Two parallel lists drift apart**. A dict keeps the pairs together.

# A list of dicts keeps each record together

## one dict per student

```
students = [  
    {"name": "Hermione", "house": "Gryffindor"},  
    {"name": "Harry", "house": "Gryffindor"},  
    {"name": "Draco", "house": "Slytherin"}  
]
```

```
# patronus: None when we do not know it
```

## reading it back

```
for student in students:  
    print(student["name"])  
Hermione  
Harry  
Draco
```

**One dict per record**, all the records in one list.

**The loop variable is a whole dict.**  
Index it by key to read a field.

**None** marks a value you simply do not have.

**This is the shape** most real data arrives in: rows of named fields.

# A loop inside a loop draws a grid

**mario.py**

```
for i in range(3):  
    for j in range(3):  
        print("#", end="")  
    print()
```

**output**

```
###  
###  
###
```

**The inner loop runs fully** on every pass of the outer one: nine prints in total.

**end=""** keeps the row on one line.

The **bare print()** ends the row and starts the next.

**Wrap the loops in a function** when you want a square of any size.

# Challenge 1: the Sorting Hat

## the task

```
# Ask for a name, then three yes/no
# questions. Count the yes answers.
# 3 -> Gryffindor, 2 -> Ravenclaw,
# 0 or 1 -> Hufflepuff.
```

## example run

```
Name? ada
Libraries? yes
Rules? no
Dogs? yes
Ada: Ravenclaw.
```

**Keep the questions in a list** and loop over them; one counter for the yes answers.

**Clean the answer.**  
`.strip().lower()` so Yes, YES and yes all count.

**Then branch. `if` / `elif` / `else`** on the counter, f-string for the verdict.

**Bonus:** if the name is Draco, print Slytherin and skip the questions.

# Sorting Hat, step by step

## sortinghat.py

```
questions = ["Libraries?", "Rules?", "Dogs?"]
name = input("Name? ").strip().title()
points = 0
for q in questions:
    answer = input(q + " ").strip().lower()
    if answer == "yes":
        points += 1
if points == 3:
    print(f"{name}: Gryffindor!")
elif points == 2:
    print(f"{name}: Ravenclaw.")
else:
    print(f"{name}: Hufflepuff.")
```

1. **Collect the questions.** A **list**, so the loop does the repeating, not you.
2. **Ask, then clean.** **.strip().lower()** before comparing, or Yes fails.
3. **Count.** **points += 1** on every yes; the counter starts at 0.
4. **Decide once.** **if / elif / else** on the total: exactly one house.
5. **Announce.** One f-string carries the name and the verdict.

# Challenge 2: Mario's pyramid

## the task

```
# Ask for a height from 1 to 8.  
# Print that many rows of #,  
# aligned to the right.  
# Row n carries n hashes.
```

## example run

```
Height? 4  
  #  
 ##  
###  
####
```

**Two loops.** The outer one counts the rows, the inner ones print the characters.

**Spaces first, then hashes.** Row 1 of 4 needs three spaces, row 4 needs none.

**end="" keeps the row open;** a bare **print()** ends it.

**Bonus:** refuse a height outside 1 to 8 and ask again.

# Pyramid, step by step

mario.py

```
height = int(input("Height? "))
for row in range(1, height + 1):
    for _ in range(height - row):
        print(" ", end="")
    for _ in range(row):
        print("#", end="")
    print()
```

height 4

```
#
##
###
####
```

1. **Read the height once.** `int()` around `input`, before any loop.
2. **One pass per row.** `range(1, height + 1)` counts 1 up to the height.
3. **Pad the left.** `height - row` spaces push the short rows to the right.
4. **Then the hashes.** `row` of them; the counter is unused, so name it `_`.
5. **Close the row.** A bare `print()` moves to the next line.

# Recap and what to do next

## What you can do now

- Write a condition: comparisons, and, or
- Pick a branch with `if`, `elif`, `else` and `match`
- Return `True` or `False` from your own function
- Repeat work with `while`, `for` and `range`, and stop it with `break`
- Keep data in a list or a dict and loop over it

## Before the next session

- Work through Problem Set 1
- Rewrite today's examples from memory, then run them
- Write one loop that you stop with `break`, and one you do not