

```
$ python test.py  
hello, prepcourse
```

Lecture 1 · Functions and Variables

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 0: Functions and Variables.
- 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

Where to write it, then six steps to a working program

1

Your tools

editor, IDE, Jupyter
Notebooks

2

First program

print and the interpreter

3

Functions, errors

arguments and tracebacks

4

Variables

input, comments,
planning

5

Strings

join, format, clean up

6

Numbers

int, float, rounding

7

Your own functions

def, parameters, return

Learning goals

By the end of the session you can

- ✓ **Run a Python file** from the terminal and read what it prints.
- ✓ **Work in an editor, an IDE or a notebook** and know what each one is for.
- ✓ **Ask for input and keep it** in a variable, then use it in the output.
- ✓ **Read an error message** and find the line that caused it.
- ✓ **Format strings and numbers** so the output looks the way you want.
- ✓ **Write your own function** with parameters and a return value.

Three hours of clicking, or four lines of Python

412 exam scans came off the scanner as DOC0001, DOC0002, DOC0003 ...

By hand

Open the folder.
Rename one file.
Repeat 411 times.

≈ 3 hours, and a few typos

In Python

```
import os
```

```
for i, name in enumerate(sorted(os.listdir("scans"))):  
    os.rename(name, f"exam_{i:03}.pdf")
```

4 seconds, and it runs the same way next time you need it.

You will not write this today. By the end of a full Python course, you will read it without help.

A short history of programming



1843

Ada Lovelace writes the first algorithm, for a machine never built.



1957

Fortran lets scientists write formulas, not machine instructions.



1972

C powers operating systems, and still sits under most of them.



1991

Python arrives: readable code first, raw speed second.



2021

Assistants start suggesting whole lines as you type.

Each step raised the level of abstraction: less machine to manage, more problem to solve.

Python spread because it is easy to read and to write

From a Christmas project in 1989 to the most used language today

- 1989** Guido van Rossum starts Python as a Christmas side project.
- 1991** First public release. The name comes from Monty Python, not the snake.
- 2008** Python 3 breaks with the past. The move takes a decade; Python 2 retires in 2020.
- Today** First in the TIOBE index, and the default language for data and AI.

Why it caught on

Readable syntax, close to English

A large standard library from day one

Free and community-run, with a package for almost everything

Source: TIOBE index, March 2026.

Machine learning is driven by Python

PyTorch, TensorFlow, JAX, scikit-learn, Hugging Face — all driven from Python

Train a classifier, start to finish

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

What runs where

You write Python

PyTorch, NumPy, pandas

C++ and CUDA on the GPU

Python itself is slow, and it rarely matters.

The heavy lifting happens in C or on the GPU. Python is where you say what you want done, which is why everyone in the field uses it.

The model writes; you decide if it is right

Vibe coding: describe what you want, accept what comes back

WHERE IT HELPS

- Boilerplate you have written a dozen times
- Syntax you forgot in a language you know
- A first draft you can react to
- Explaining an error message in plain words

WHERE IT LEAVES YOU ALONE

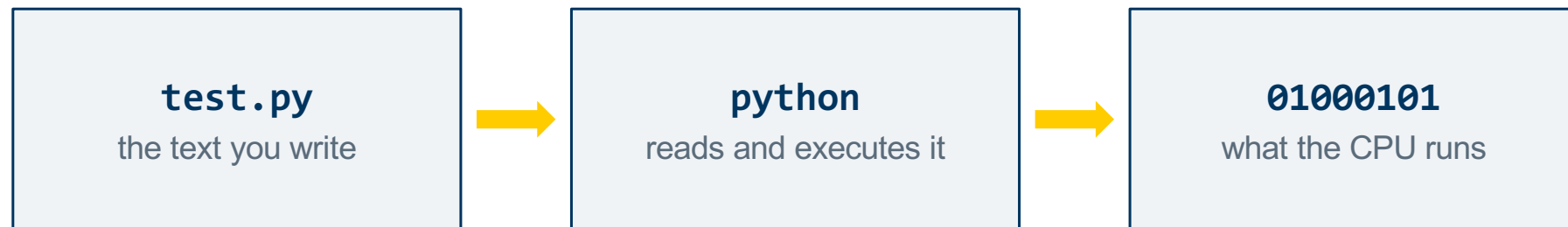
- Knowing what to ask for in the first place
- Catching an answer that looks right and is wrong
- Owning the result when your name is on it
- Building skill, which comes from writing code yourself

You cannot check an answer in a language you cannot read. This course teaches you to read it, test it, and understand where and when it is wrong. That judgment is the part the model cannot do for you.

Your code is just text; the interpreter runs it

Python is a language you can read out loud

- A program is plain text, written in an IDE and saved as `test.py`
- `python test.py` hands that file to the interpreter
- The interpreter reads it line by line and does what it says



The tools changed more than the language did

Three places to write the same file, in the order they appeared

THE BASICS

Editor + terminal

Write the file in any editor, run it from the terminal. One file, one command.



THE WORKBENCH

IDE

Editor, terminal and error marks in one window. Same file, same command, help as you type.



THE SKETCHPAD

Jupyter notebook

Code in cells, output under each one. Good for trying things; programs still live in a .py file.

All three run the same Python. Learn the file and the command first; the tools only save you typing.

The same two lines, in four places

The same two lines of Python, live in each of the four windows

Terminal

```
$ python test.py
What's your name? Nils
hello, Nils
```

Text editor

test.py

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

IDE

test.py

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

```
$ python test.py
hello, Nils
```

Jupyter notebook

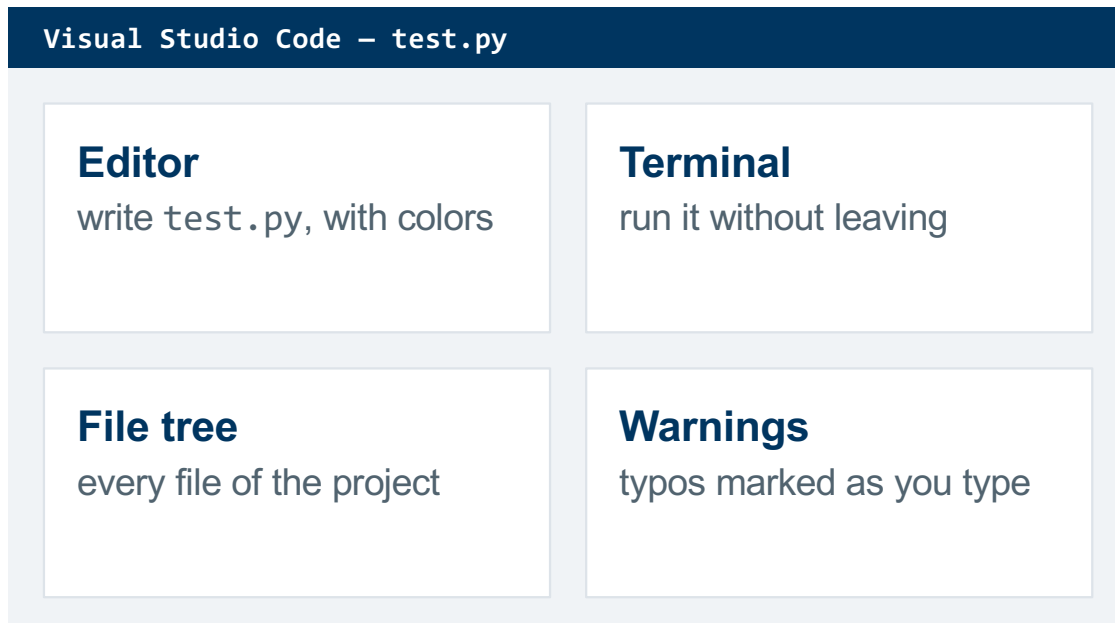
test.ipynb

```
In [1]: name = input("Your name? ")
        print("hello,", name)
```

```
Your name? Nils
hello, Nils
```

An IDE bundles editor, terminal and helpers

Integrated development environment — everything in one window



IDE means integrated.

Editor, terminal, file tree and warnings in one window.

Nothing new runs. The IDE calls the same python command.

Mistakes surface earlier. A red mark beats a traceback afterwards.

We use VS Code. Free, cross-platform, the common choice for Python.

Part 1

Your first program

 1 · Your first program

 2 · Strings

 3 · Numbers

 4 · Your own functions

`print("hello, world")` is a whole program

test.py

```
print("hello, world")
```

terminal

```
$ python test.py  
hello, world
```

One line is enough. No imports, no main, no semicolons.

print is a function. It displays whatever you hand it.

"hello, world" is the argument. The text in quotes is what gets printed.

You run it yourself. `python test.py` in the terminal.

A function acts; arguments are its input

A function is an action the language already knows

- You call it by name and put its arguments in parentheses
- `print` writes to the screen: that visible result is its side effect
- Other functions hand a value back instead, called a return value

```
print("hello, world")
```

the function

the argument

Change the argument and the output changes. The function stays the same.

Errors happen, and the message tells you where

test.py

```
print("hello, world"
```

terminal

```
$ python test.py
  File "test.py", line 1
    print("hello, world"
      ^
SyntaxError: '(' was never closed
```

Read the last line first. It names the problem.

Then read the line number. It says where Python gave up.

A missing bracket is the most common error. You will make it often.

Debugging is the work, not a sign you cannot do this.

using inputs and what's the problem here?

test.py

```
input("What's your name? ")  
print("hello, world")
```

terminal

```
$ python test.py  
What's your name? Nils  
hello, world
```

input prints a prompt and waits. It hands back whatever was typed.

Nothing catches the answer, so Python throws it away.

The answer needs somewhere to live. That is a variable.

A variable is a named container for a value

test.py

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

terminal

```
$ python test.py
What's your name? Nils
hello,
Nils
```

= does not mean equals. It stores the value on the right.

The right side runs first. input asks, then the answer lands in name.

print(name) prints the value.
print("name") prints the word name.

Two calls, two lines of output.
Joining them into one line comes next.

Working code is only half the job

1

Plan

Write the steps in plain words before typing any Python.

2

Write

Turn one step at a time into code, then run the file.

3

Check

Compare the output with the plan and fix what differs.

You will read your program far more often than you write it.

A comment is a line Python ignores

weak — repeats the code

```
# Call input and store the result  
name = input("What's your name? ")
```

useful — says why

```
# The greeting needs a name, so ask first  
name = input("What's your name? ")
```

Python skips the # and the rest of the line.

Comments are for people.

One note per step is enough. A comment on every line is noise.

Plan in comments, then fill in the code

step 1 — write the plan

```
# Ask the user for their name  
# Say hello  
# Print the name
```

step 2 — fill in the code

```
# Ask the user for their name  
name = input("What's your name? ")  
  
# Say hello and print the name  
print("hello,")  
print(name)
```

Start with the to-do list. Three comments, no code yet.

Fill in one step at a time. Run the file after each step.

The comments stay. They label the finished program.

Blank lines are free. Use them to group steps.

Part 2

Strings

1 · Your first program

2 · Strings

3 · Numbers

4 · Your own functions

Three ways to join text, one you should use

glue with +

```
print("hello, " + name)
```

two arguments

```
print("hello,", name)
```

f-string — use this one

```
print(f"hello, {name}")
```

All three print hello, Nils. The difference is how they read.

+ glues strings together. You have to add the space yourself.

A comma passes two arguments. print puts a space between them.

An f-string holds the value inline. The f makes {name} a slot to fill.

Parameters have defaults you can override

two prints, two lines

```
print("hello,")  
print(name)  
  
hello,  
Nils
```

override the end parameter

```
print("hello,", end=" ")  
print(name)  
  
hello, Nils
```

print adds a line break at the end. This behavior is defined via its **end parameter**.

Set end to change that. With `end=" "` the next line of output stays on the same line.

A parameter is a named option. It has a value already; pass one only to override it.

The docs list them all. Look up `print` to see the rest.

How to get help? Look up the documentation.

`docs.python.org` — the entry for print

```
print(*objects, sep=' ', end='\n',  
      file=None, flush=False)  
Prints objects to the text stream file,  
separated by sep and followed by end.
```

or ask Python itself

```
$ python  
>>> help(print)  
>>> help(str.strip)
```

`docs.python.org` is the official manual, written by the people who build Python.

The Library Reference is the part you want; built-in functions come first.

Read the signature. It names every parameter and its default value.

`help()` works offline. Same text, and it matches the version you run.

Clean the input first: strip() and title()

clean it up, one step at a time

```
name = input("What's your name? ")
name = name.strip()
name = name.title()
print(f"hello, {name}")
```

the same thing, chained

```
name = input("What's your name? ")
name = name.strip().title()
print(f"hello, {name}")
```

Users type stray spaces. `nils` is not the same string as `nils`.

`strip()` removes the whitespace

`title()` capitalises each word.

Methods are functions on a value. You call them with a dot after the string.

Each one returns a new string. So you can chain them left to right.

split() cuts one string into several

first name, last name

```
name = input("What's your name? ").strip()
first, last = name.title().split(" ")
print(f"hello, {first}")
```

output

```
What's your name?  ada lovelace
hello, Ada
```

split(" ") cuts at every space. Two words in, two strings out.

Two names on the left. Python assigns the pieces in order.

The counts must match. One word alone raises a ValueError.

strip() first, or a trailing space becomes an extra piece.

Part 3

Numbers

1 · Your first program

2 · Strings

3 · Numbers

4 · Your own functions

input gives you text, so 1 + 1 becomes 11

calculator.py

```
x = input("What's x? ")    # What's x? 1
y = input("What's y? ")    # What's y? 2
z = x + y
print(z)                   # 12
```

the fix: cast to int

```
x = int(input("What's x? "))
print(x + y)               # 3
```

+ on two strings glues them. "1" + "2" is the text 12, not 3.

input always returns text. Whatever is typed arrives as a str.

int() converts text to a number. That is casting.

Functions nest. The inner call runs first.

Fewer variables. print(int(x) + int(y)) drops z.

Operators: + - * / and % for the remainder

the five you need today

```
print(7 + 3)    # 10
print(7 - 3)    # 4
print(7 * 3)    # 21
print(7 / 3)    # 2.3333333333333335
print(7 % 3)    # 1
```

% answers: what is left over?

```
print(10 % 2)   # 0, so even
print(11 % 2)   # 1, so odd
```

Four are the familiar ones. Same meanings as on paper.

/ always gives a float. 6 / 3 is 2.0, not 2.

% is the remainder. Use it for even, odd, and every nth item.

Precedence is the usual one. * and / happen before + and -.

Interactive mode: python with no file name

terminal

```
$ python
>>> 1 + 1
2
>>> name = "nils"
>>> name.title()
'Nils'
>>> exit()
$
```

Run python on its own. The >>> prompt appears instead of a file.

One line in, one answer out. Nothing is saved, so you cannot damage a file.

Use interactive mode to check a detail. What does % do again? Try it.

exit() closes it. Real programs still live in .py files.

Floats are approximate, so format the output

calculator.py

```
x = float(input("x? ")) # x? 2
y = float(input("y? ")) # y? 3
print(x / y)             # 0.6666666666666666
```

round the number, or format the text

```
print(round(x + y))      # 5, a whole number
print(round(x / y, 2))   # 0.67
print(f"{x / y:.1f}")    # 0.7
print(f"{x / y:.2f}")    # 0.67
print(f"{x / y:.3f}")    # 0.667
print(f"{1000000:,}")    # 1,000,000

print(round(2.5, 2))     # 2.5, zero lost
print(f"{2.5:.2f}")      # 2.50, two digits
```

A float has a decimal point.

`float()` casts text to one, like `int()` does.

Division rarely comes out round.

Sixteen digits are binary, showing its limits.

`round(z, 2)` changes the number.

Use it when the value itself matters.

`f"{z:.2f}"` changes the display. Use it when only the printout matters.

Part 4

Writing your own functions

1 · Your first program

2 · Strings

3 · Numbers

4 · Your own functions

def names a block; indentation defines it

test.py

```
def hello():  
    print("hello")  
  
name = input("Your name? ")  
hello()  
print(name)
```

output

```
Your name? Nils  
hello  
Nils
```

def creates a function. The name, brackets, then a colon.

The indented lines are the body. In Python that indent is syntax, not style.

Defining a function runs nothing. `hello()` is what actually calls it.

Four spaces per level. Never mix tabs and spaces in one file.

Parameters carry values into a function

test.py

```
def hello(to="world"):
    print("hello,", to)

name = input("Your name? ")
hello(name)
hello()
```

output

```
hello, Nils
hello, world
```

to is a parameter. A name the function uses for whatever arrives.

hello(name) passes a value in.
Inside the function, to holds it.

The = gives a default. hello() with no argument falls back to world.

Names are local. to only exists inside hello.

main() at the bottom keeps programs readable

test.py

```
def main():  
    name = input("Your name? ")  
    hello(name)  
  
def hello(to="world"):  
    print("hello,", to)  
  
main()
```

Python reads the file top to bottom. Defining a function does not run it.

Without the last line, nothing happens. `main()` is the switch that starts the program.

Put the story first. `main()` reads like a summary; details sit below.

This shape is a convention. Almost every Python file looks like this.

return sends a value back to the caller

calculator.py

```
def main():  
    x = int(input("x? "))  
    print("x squared is", square(x))  
  
def square(n):  
    return n * n  
  
main()
```

output

```
x? 5  
x squared is 25
```

print shows a value. return hands it back to whoever asked.

square(x) becomes 25. The call is replaced by the value it returns.

No return means None. The function still runs; it just answers nothing.

This is how programs compose. Small functions that answer, used inside others.

Recap and outlook

What you can do now

- Write and run a program: `print`, `input`, variables
- Work with text: `f-strings`, `strip`, `title`, escaped quotes
- Work with numbers: `int`, `float`, operators, rounding
- Write your own functions: `def`, parameters, defaults, `return`
- Read an error message and fix the line it names

Before the next session

- Retype every example from today by hand, without copying
- Change an example and predict the output before running it
- Work through the exercises
- Keep a file of what broke and how you fixed it