PYTHON

BEGINNER WORKBOOK

Learn Python by Solving Real Problems

9 UNITS · FLOWCHARTS · 150+ EXERCISES · CAPSTONE PROJECT · ANSWER KEY

```python
def solve_problem(steps):
    """From confusion to clarity — one step at a time."""
    algorithm = Flowchart(steps)
    for step in algorithm:
        think(step)  # understand first
        write(step)  # then code
        test(step)  # always test
    return confidence + skill

# 9 units · algorithms · data types · functions
# error handling · files · capstone project  ↓
solve_problem("python_fundamentals")
```

Independent Learning Series

Practical Python Workbook Series

FIRST EDITION

Python Beginner Workbook

Practical Python Workbook Series

First Edition, 2026

Printed and distributed independently.

A Note From the Author

I still remember the exact moment I fell in love with programming. I was seventeen, sitting in a cold computer lab after school, trying to make a simple loop count to ten. It ran forever. The fan on the PC screamed. I had to yank the power cable from the wall. My teacher came over, glanced at my screen, and said — with the patience of a saint — *“You forgot the increment.”* Two characters. That was it.

I share that story because I want you to know something important before you open this book in earnest: **everyone starts confused**. Everyone writes bugs that make them feel stupid. Everyone has that moment staring at a blinking cursor thinking “this is impossible.” It is not. It just takes practice — and a guide that does not pretend it is easy when it is not.

This workbook was written for one reader: someone who has never programmed before and wants to actually *understand* what they are doing, not just copy code from a screen. Every concept is explained from first principles. Every code example is something you can type, run, break, and fix. Every exercise is designed to make you think.

How This Book Works

There are 9 units. Each builds on the last. Unit 1 teaches you to think like a programmer before you write a single line of Python. Unit 9 brings everything together in a capstone project you will actually be proud of. Each unit contains: concept explanations, worked examples with full code, flowcharts and diagrams, key terms, and exercises. Do not skip ahead — the foundations matter more than they appear.

Three rules I would ask of you:

- **Type every example yourself.** Your fingers need to learn the syntax as much as your brain does. Copying and pasting is cheating yourself.
- **Read the error messages.** Python's errors are remarkably helpful once you learn to read them. Most beginners skip past them in panic. I will teach you to read them calmly.
- **Break things deliberately.** Once an example works, change something. Remove a colon. Swap indentation. Use the wrong operator. Understanding why things break is how you understand why things work.

Programming is one of the most transferable skills you can learn. Not because it guarantees a job — though it might — but because it teaches you to decompose complex problems, reason precisely, and create something from nothing but logic and a keyboard. That is genuinely remarkable. Now let us build something.

— Fatih Arica

How to Use This Workbook

This workbook is designed for beginners who want to learn Python through **practice rather than long theoretical explanations**. If you have never written a line of code before, you are in the right place. If you have tried other Python tutorials and found them too abstract, this workbook was written specifically for you.

Each unit in this workbook contains:

- **Short concept explanations** — just enough theory to understand what you are doing and why, without drowning you in detail.
- **Flowcharts and diagrams** — visual representations of algorithms and logic structures that make abstract ideas concrete.
- **Worked examples** — complete, runnable Python programs with line-by-line explanations.
- **Exercises** — hands-on problems that reinforce each concept. Always attempt these before checking the answer key.
- **Mini projects** — longer challenges that combine multiple concepts from the unit.

How to get the most out of this workbook:

- Type every code example yourself. Do not copy and paste. Your hands need to learn the syntax alongside your brain.
- Read error messages carefully. Python’s error messages are informative once you learn to interpret them. Unit 1 will teach you how.
- Work through the units in order. Each unit builds on the previous one. The foundations in Unit 1 matter more than they appear.
- Attempt every exercise before checking answers. Struggle is where learning happens. Looking up the answer immediately robs you of that.

You will need Python 3.8 or later installed on your computer. Unit 2 walks you through the installation process step by step. A plain text editor or the free VS Code editor is recommended. No prior programming experience is required — only curiosity and patience.

Table of Contents

Unit 01 — Problem Solving & Algorithms

Unit 02 — Introduction to Programming

Unit 03 — Data Types & Variables

Unit 04 — Decision & Loop Structures

9.7 What Comes Next

UNIT 01

Problem Solving & Algorithms

Think before you type — the mental toolkit every programmer needs

- ✔ Define a computing problem precisely, identifying inputs, outputs, and constraints
- ✔ Apply the six-step problem-solving process to any programming challenge
- ✔ Write algorithms in plain English and structured pseudocode
- ✔ Draw flowcharts using standard symbols (terminal, process, decision, I/O)
- ✔ Use all five Python operator families and apply precedence rules correctly
- ✔ Read a Python error message without panicking and locate the root cause

1.1 What Is a Problem?

Let us start with a question that sounds obvious: what *is* a problem? In everyday life we use the word loosely. “My phone is a problem.” “Traffic is a problem.” But in computing we need to be precise. A **computing problem** has exactly three parts:

Component	Description	Example: Navigation App
Current state	Where things are right now	User is at home (lat/lon A)
Goal state	Where we want things to be	User needs to reach the office (lat/lon B)
Gap	The difference we must bridge	An optimal route between A and B

Every program ever written exists to bridge a gap between a current state and a goal state. A spell-checker bridges the gap between “this word might be wrong” and “I know the correct spelling.” A payroll system bridges the gap between “hours worked” and “correct salary payments.”

Understanding the problem before writing code is the single most important habit you can develop. Researchers studying expert programmers consistently find that experts spend proportionally more time on problem analysis than beginners — and significantly less time debugging. The investment pays off immediately.

Real Talk: The Debugger Who Never Read the Brief

A student on my course once spent four hours debugging a temperature converter. The code was technically correct. The problem was that the brief asked for Celsius to Kelvin, not Celsius to Fahrenheit. He had solved the wrong problem perfectly. Define the problem first. Read the brief twice. The code comes last.

1.2 The Six-Step Problem-Solving Method

Professional software engineers do not sit down and start typing. They follow a structured process. Here is the method I use and teach:

Step	Name	What You Actually Do
1	Understand	Restate the problem in your own words. What are the exact inputs? What must the output be? What counts as success? Write this down.
2	Identify Constraints	What are the rules, limits, edge cases? What happens if the input is zero? Negative? Empty? Very large?
3	Plan (Pseudocode)	Write the steps in plain structured English. No code yet. Walk through your plan with an example by hand.
4	Check the Plan	Trace through your pseudocode with real example data. Does it give the right answer? What about edge cases?
5	Code	Translate your verified plan into Python. One step at a time. Test each piece before adding the next.
6	Test & Debug	Try valid inputs, invalid inputs, boundary values, and extreme cases. Fix what breaks. Document what you found.

■ Pro Tip

Print this table. Stick it above your monitor. Every professional developer I know follows this process, consciously or not. Those who skip to step 5 consistently spend twice as long in step 6.

Worked Application: ATM Cash Withdrawal

Let us apply all six steps to a concrete problem: designing the logic for an ATM cash withdrawal.

Step	Application to ATM Problem
1. Understand	Input: card PIN, requested amount. Output: cash dispensed or error message. Success: correct amount given, balance updated.
2. Constraints	PIN must match. Amount must be positive. Amount must be multiple of 10. Amount must not exceed balance. Amount must not exceed daily limit (e.g. 500).
3. Pseudocode	IF pin correct AND amount valid AND amount <= balance AND amount <= daily_limit THEN dispense(amount), update_balance(amount) ELSE show_error()
4. Check	Trace with PIN=1234 (correct), amount=200, balance=350, daily=500. Each check passes. Result: dispense 200. Check with amount=600 (over daily limit): blocked. Correct.
5. Code	Translate to Python if/elif/else. See example below.
6. Test	Test: wrong PIN, correct PIN, zero amount, negative amount, non-multiple of 10, over balance, over daily limit, exact balance.

atm_withdrawal.py

```python
# ATM Withdrawal Logic
def process_withdrawal(stored_pin, entered_pin, amount, balance, daily_limit):
    if entered_pin != stored_pin:
        return False, 'Incorrect PIN'
    if amount <= 0:
        return False, 'Amount must be positive'
    if amount % 10 != 0:
        return False, 'Amount must be a multiple of 10'
    if amount > balance:
        return False, 'Insufficient funds'
    if amount > daily_limit:
        return False, f'Exceeds daily limit of {daily_limit}'
    new_balance = balance - amount
    return True, f'Dispensing {amount}. New balance: {new_balance}'

# Test cases
print(process_withdrawal(1234, 1234, 200, 350, 500))  # OK
print(process_withdrawal(1234, 9999, 200, 350, 500))  # Wrong PIN
print(process_withdrawal(1234, 1234, 600, 350, 500))  # Over daily
print(process_withdrawal(1234, 1234,  25, 350, 500))  # Not mult of 10
```

1.3 Operators and Precedence

Python has five families of operators. You need all of them to write real programs. Here is a complete reference:

Family	Operators	Purpose & Notes
Arithmetic	+ - * / // % **	Maths. / always returns float. // floor divides (rounds down). % is modulo (remainder). ** is power.
Assignment	= += -= *= /= //= %= **=	Store and update. x += 3 is shorthand for x = x + 3.
Comparison	== != > < >= <=	Always returns True or False. == tests equality; = assigns. Easy to confuse.
Logical	and or not	Combine conditions. Short-circuits: and stops at first False, or stops at first True.
Identity	is is not	Tests if two variables point to the exact same object in memory. Use == for value equality, is for identity (e.g. is None).

Operator Precedence — Evaluation Order

When an expression contains multiple operators, Python evaluates them in a fixed order. This mirrors mathematical BODMAS/PEMDAS with extra rules for logical operators:

Priority	Operator(s)	Name
1 (highest)	( )	Parentheses — always first
2	**	Exponentiation (right-associative)
3	+ - (unary)	Positive / negative sign
4	* / // %	Multiplication, division, floor div, modulo
5	+ -	Addition, subtraction
6	== != > < >= <=	Comparison operators
7	not	Logical NOT
8	and	Logical AND
9 (lowest)	or	Logical OR

precedence.py

```
# Precedence in practice &#8212; predict before running
print(2 + 3 * 4)         # 14  (* before +)
print((2 + 3) * 4)       # 20  (parentheses first)
print(2 ** 3 ** 2)       # 512 (** is right-associative: 3**2=9, 2**9=512)
print(10 % 3 + 10 // 3)  # 4   (1 + 3)
print(not 5 > 3)         # False (5>3 is True, not True is False)

# The classic beginner trap
x = 5
if x == 3 or 4:          # WRONG! This is always True
    print('matches')     # Python reads: (x==3) or (4) &#8212; 4 is truthy

if x == 3 or x == 4:    # CORRECT
    print('matches')

# Chained comparisons (unique to Python)
age = 25
if 18 <= age < 65:       # More readable than age >= 18 and age < 65
    print('Working age')
```

1.4 Algorithms and Pseudocode

An **algorithm** is a finite, unambiguous sequence of steps that solves a problem and always terminates. The word comes from the name of 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi — so algorithms predate computers by over a thousand years. Three properties are required:

- **Finiteness:** the algorithm must end after a finite number of steps. An infinite loop is not an algorithm.
- **Definiteness:** every step must be precisely defined. “Sort of add them” is not a valid step.
- **Effectiveness:** each step must be achievable with available resources in finite time.

Pseudocode is algorithm expressed in structured English — precise enough to translate directly into code, but free of syntax rules. There is no single correct format; clarity is the only standard.

Pseudocode Convention	Meaning
INPUT variable	Read a value from the user
OUTPUT expression	Display a value to the user
variable ← expression	Assign a value (some styles use =)
IF condition THEN ... ELSE ... ENDIF	Conditional execution
FOR variable FROM x TO y ... ENDFOR	Count-controlled loop
WHILE condition ... ENDWHILE	Condition-controlled loop
CALL function(args)	Call a named procedure
RETURN value	Return a result from a procedure

Pseudocode Example: Find Maximum of Three Numbers

```
INPUT a, b, c
max ← a
IF b > max THEN
    max ← b
ENDIF
IF c > max THEN
    max ← c
ENDIF
OUTPUT max
```

find_maximum.py

```python
# The same algorithm in Python
a = float(input('Enter first number: '))
b = float(input('Enter second number: '))
c = float(input('Enter third number: '))

maximum = a
if b > maximum:
    maximum = b
if c > maximum:
    maximum = c
```

```
print(f'The maximum is: {maximum}')

# Python's built-in equivalent (good to know):
print(max(a, b, c))
```

1.5 Flowcharts

A **flowchart** is a visual representation of an algorithm using standardised shapes connected by arrows. Flowcharts are used to plan programs, communicate logic to non-programmers, and document existing systems. They are especially powerful for showing decision branches — which path does the program take, and when?

Symbol	Shape	Used For
Terminal	Rounded rectangle (oval)	Start or End of the algorithm
Process	Rectangle	Any action: assignment, calculation, output
Decision	Diamond	A yes/no question — two output arrows (Yes/No)
Input/Output	Parallelogram	Reading input from user or writing output
Arrow	Line with arrowhead	Direction of control flow between steps
Connector	Small circle	Continue flow on a new page or distant location

Flowchart: Grade Classifier

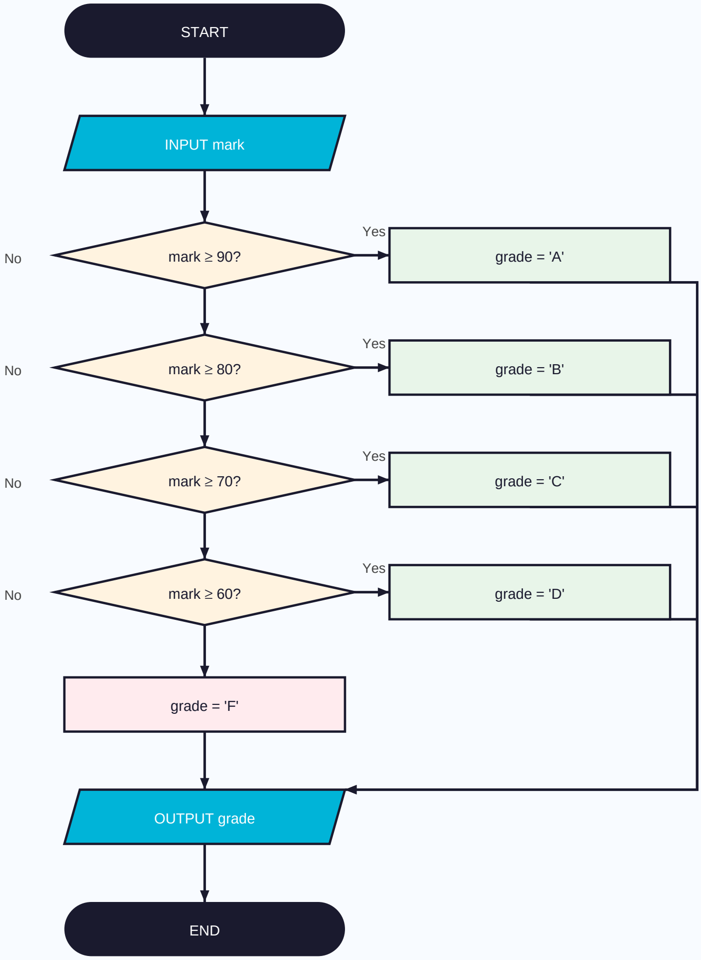

Figure 1.1. Grade Classification Algorithm

grade_classifier.py

```python
# The same logic in Python
mark = int(input('Enter your mark (0-100): '))

if mark >= 90:   grade = 'A'
elif mark >= 80: grade = 'B'
elif mark >= 70: grade = 'C'
elif mark >= 60: grade = 'D'
else:            grade = 'F'

print(f'Your grade is: {grade}')
```

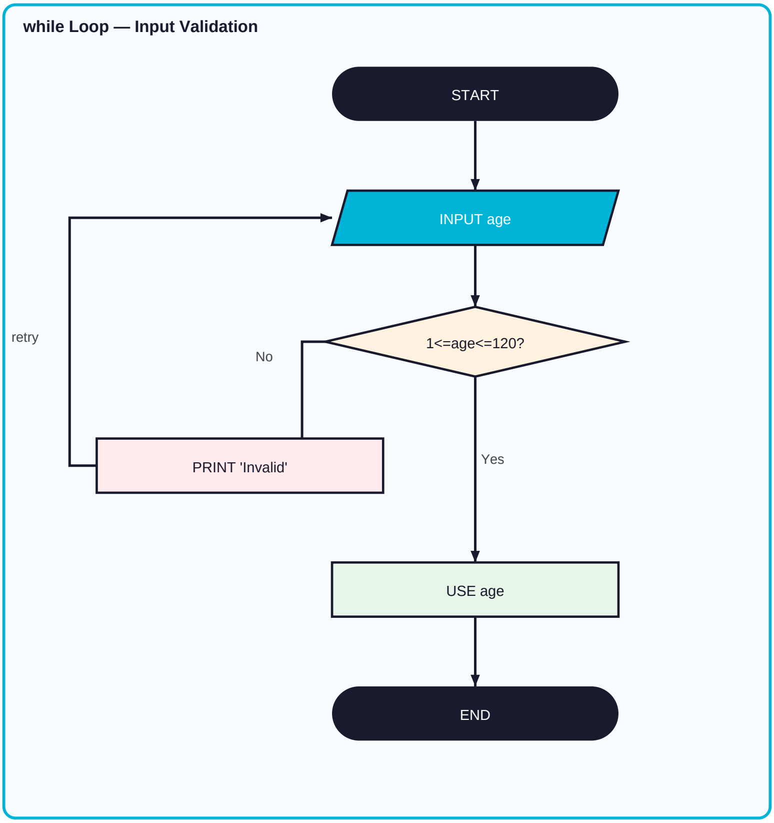

Figure 4.3. while Loop — Input Validation Pattern

Figure 1.2. Input Validation with while Loop

1.6 Debugging: Reading Error Messages

Here is professional truth: a significant portion of programming is debugging. Not because you are bad at it — because everyone debugs. The difference between junior and senior developers is not the frequency of bugs; it is the speed and calm of the debugging process.

Error Type	When It Occurs	Example	Primary Fix Strategy
Syntax Error	Before code runs — Python cannot parse it	Missing colon, unmatched parenthesis	Python points to the line. Read the message carefully.
Runtime Error	While code is running	Division by zero, wrong type, missing key	Traceback shows the exact line. Check variable values with print().
Logic Error	Code runs but gives wrong output	Using + when you meant *, off-by-one in loop	Add print() statements. Trace values step by step. Use a debugger.

Python's traceback is your best friend. It tells you: (1) which file failed, (2) which line failed, (3) the call stack (which functions called which), and (4) the exception type and message. Most beginners read only the last line. Read *all* of it.

debugging.py

```
# Example traceback &#8212; how to read it
# Traceback (most recent call last):
#   File 'program.py', line 12, in <module>     <- which file, which line
#     result = calculate(data)                   <- what was executing
#   File 'program.py', line 7, in calculate     <- call stack
#     return total / count                       <- the failing line
# ZeroDivisionError: division by zero           <- exception type + message

# Diagnostic: add print() to trace values
def calculate(data):
    total = sum(data)
    count = len(data)
    print(f'DEBUG: total={total}, count={count}')  # add temporarily
    return total / count                            # fails if count == 0

# Fix: handle the edge case
def calculate_safe(data):
    if not data:              # empty list
        return 0
    return sum(data) / len(data)
```

■ Watch Out

Never name your variables after Python built-ins: list, str, int, float, input, print, sum, max, min, type, range. Doing so silently overwrites the built-in, causing mysterious errors that are very hard to find. Use descriptive names: name_list, total_sum, user_input.

1.7 Worked Examples

Example 1: Swap Two Variables

swap.py

```python
# Method 1: temporary variable (works in all languages)
a, b = 10, 25
temp = a
a = b
b = temp
print(a, b)   # 25 10

# Method 2: Python's elegant tuple swap
a, b = 10, 25
a, b = b, a   # right side evaluated fully before assignment
print(a, b)   # 25 10
```

Example 2: FizzBuzz (the universal beginner benchmark)

fizzbuzz.py

```python
# Print 1-50. Fizz if divisible by 3, Buzz if by 5, FizzBuzz if both.
# KEY: check the combined case FIRST
for n in range(1, 51):
    if n % 3 == 0 and n % 5 == 0:
        print('FizzBuzz')
    elif n % 3 == 0:
        print('Fizz')
    elif n % 5 == 0:
        print('Buzz')
    else:
        print(n)
```

Example 3: Leap Year Checker

A leap year is divisible by 4, EXCEPT centuries (divisible by 100), UNLESS also divisible by 400. This is the perfect multi-condition algorithm exercise.

Flowchart: Leap Year Checker

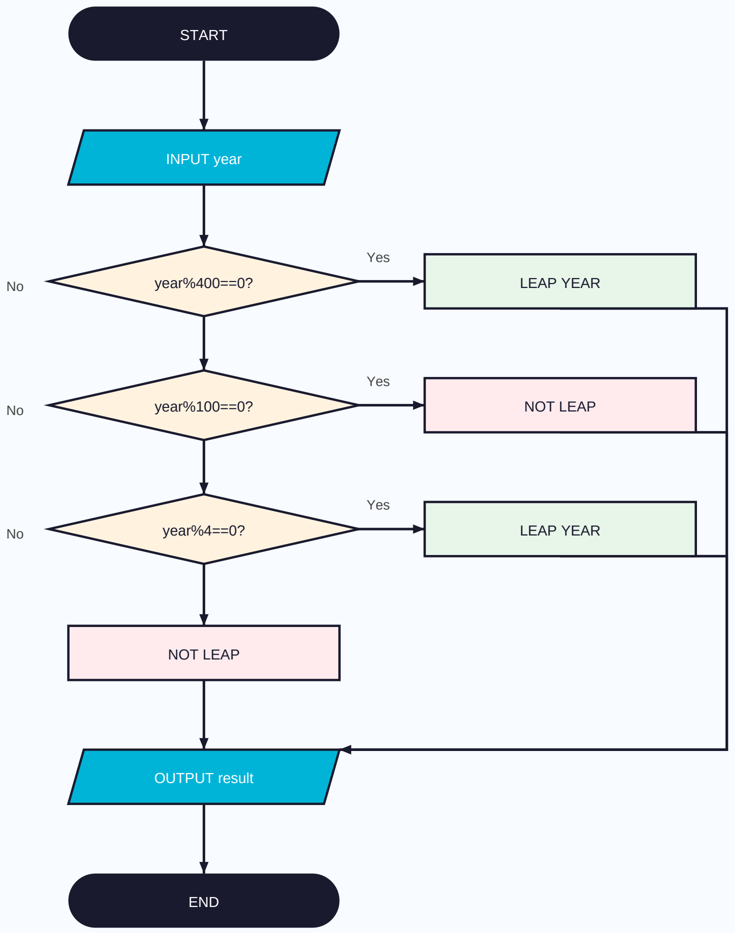

Figure 1.3. Leap Year Decision Algorithm

leap_year.py

```python
def is_leap_year(year):
    if year % 400 == 0:
        return True
    if year % 100 == 0:
        return False
    if year % 4 == 0:
        return True
    return False

# Test all the interesting cases
test_years = [2000, 1900, 2024, 2023, 1600, 1700]
for y in test_years:
    status = 'LEAP' if is_leap_year(y) else 'not leap'
    print(f'{y}: {status}')
# 2000: LEAP (div by 400)
# 1900: not leap (div by 100, not 400)
# 2024: LEAP (div by 4, not 100)
# 2023: not leap
```

1.8 Key Terms & Exercises

Term	Definition
Algorithm	A finite, unambiguous, effective sequence of steps that solves a problem and always terminates.
Pseudocode	Algorithm written in structured English — precise but free of programming syntax.
Flowchart	A visual diagram of an algorithm using standardised shapes (terminal, process, decision, I/O).
Input	Data provided to a program for processing (e.g. user keyboard entry, file contents).
Output	The result produced by a program (e.g. displayed text, written file, returned value).
Operator precedence	The order in which Python evaluates operators in an expression when no parentheses specify otherwise.
Debugging	The process of finding and fixing errors in a program.
Traceback	Python's error report showing the sequence of function calls that led to an exception.