369 TESLA | KALVIYOGI

UNIT VII

Python Programming



Basics · Libraries · Conditionals · Functions
Strings · Lists · Tuples & Dictionaries

```
def greet(name):
return "Hi " + name

for n in ["Asha", "Ravi"]:
print(greet(n))

# Hi Asha
# Hi Ravi
```

Kalviyogi Nagarajan

EDUNIVRA PVT LTD

UNIT VII

PYTHON PROGRAMMING

Complete Visual Notes with Real Program Screens, Output Windows and 3D Infographics

Kalviyogi Nagarajan

EDUNIVRA PVT LTD

Python Programming — Unit VII
Kalviyogi Nagarajan · 369 TESLA | KALVIYOGI

Published by EDUNIVRA PVT LTD.

This edition is a redesigned and edited version of the author's Unit VII notes. Every program screen in this book was run with Python 3.13; the output windows show the real output produced by the interpreter. Technical statements were checked and corrected where needed, and the figures are redrawn as vector 3D illustrations.

For the TRB Computer Instructor examination, every chapter ends with **Exam Theory Notes**: key facts plus predict-the-output items whose results were verified by running them. Appendices A–J add quick-reference tables, a predict-the-output bank and a last-day revision.

Product names (Python, NumPy, Pandas, Matplotlib, SciPy, Windows, Linux, macOS) belong to their owners and are used only to identify real software.

Contents

Python Basics 21

Chapter 1 · Introduction to Python 22
- 1.1 Meaning of Python 22
- 1.2 Features of Python 23
- 1.3 Applications of Python 24
- 1.4 Python as an Interpreted Language 25
- 1.5 Python Program Execution Process 25
- 1.6 Exam Theory Notes 26

Chapter 2 · The Python Interpreter 28
- 2.1 Meaning of Interpreter 28
- 2.2 Role of the Python Interpreter 29
- 2.3 Interpreter versus Compiler 29
- 2.4 Installing and Opening the Python Interpreter 30
- 2.5 Writing the First Python Command 31
- 2.6 Exam Theory Notes 32

Chapter 3 · Interactive Mode and Script Mode 34
- 3.1 Meaning of Interactive Mode 34
- 3.2 Using the Python Shell and IDLE 35
- 3.3 Meaning of Script Mode 35
- 3.4 Writing and Saving Python Programs 36
- 3.5 Interactive Mode versus Script Mode 37
- 3.6 Exam Theory Notes 37

Chapter 4 · Errors and Debugging 39
- 4.1 Meaning of Error 39
- 4.2 Syntax Errors 40
- 4.3 Runtime Errors 41
- 4.4 Logical Errors 42
- 4.5 Error Messages and Tracebacks 42
- 4.6 Simple Debugging Techniques 43
- 4.7 Exam Theory Notes 44

Chapter 5 · Values and Data Types 45
- 5.1 Meaning of Value 45
- 5.2 Integer Type 46
- 5.3 Float Type 47
- 5.4 Boolean Type 47
- 5.5 String Type 48
- 5.6 List Type 48
- 5.7 Type Conversion Basics 48
- 5.8 Checking Type using type() 49
- 5.9 Exam Theory Notes 50

Chapter 6 · Variables in Python 51
- 6.1 Meaning of Variable 51
- 6.2 Need for Variables 52
- 6.3 Variable Naming Rules 53
- 6.4 Assigning Values to Variables 53
- 6.5 Multiple Assignment 54

6.6 Dynamic Typing in Python 55
6.7 Common Variable Naming Mistakes 55
6.8 Exam Theory Notes 56

Chapter 7 · Expressions, Statements, Precedence and Comments 57
7.1 Meaning of Expression 57
7.2 Meaning of Statement 58
7.3 Difference between Expression and Statement 58
7.4 Evaluation of Expressions 59
7.5 Simple Python Statements 59
7.6 Precedence of Operators — Complete Table 60
7.7 Tuple Assignment 61
7.8 Comments 61
7.9 Exam Theory Notes 61

Python Libraries 63

Chapter 8 · Introduction to Python Libraries 64
8.1 Meaning of Python Library 64
8.2 Need for Python Libraries 65
8.3 Built-in Libraries and External Libraries 65
8.4 Package, Module and Library Difference 66
8.5 Installing Libraries using pip 67
8.6 Importing Libraries in Python 67
8.7 Library Alias Name using as 68
8.8 Commonly Used Python Libraries 69
8.9 Exam Theory Notes 69

Chapter 9 · NumPy Library 71
9.1 Introduction to NumPy 71
9.2 Need for NumPy 72
9.3 NumPy Array Meaning 72
9.4 Creating NumPy Arrays 73
9.5 One-Dimensional and Two-Dimensional Arrays 74
9.6 Array Indexing and Slicing 74
9.7 Array Shape, Size and Data Type 75
9.8 NumPy Mathematical Operations 76
9.9 NumPy Statistical Functions 76
9.10 Difference between Python List and NumPy Array 77
9.11 Exam Theory Notes 77

Chapter 10 · Pandas Library 79
10.1 Introduction to Pandas 79
10.2 Need for Pandas 80
10.3 Series in Pandas 80
10.4 DataFrame in Pandas 81
10.5 Creating a DataFrame 82
10.6 Reading Data from CSV Files 82
10.7 Viewing Data using head() and tail() 83
10.8 Selecting Rows and Columns 83
10.9 Handling Missing Values 84
10.10 Basic Data Analysis using Pandas 84
10.11 Exam Theory Notes 85

Chapter 11 · Matplotlib Library 87

11.1 Introduction to Matplotlib 87
11.2 Need for Data Visualization 88
11.3 Line Chart 89
11.4 Bar Chart 90
11.5 Pie Chart 91
11.6 Scatter Plot 92
11.7 Histogram 92
11.8 Adding Title, Labels and Legends 93
11.9 Customizing Graphs 94
11.10 Importance of Visualization in Data Analysis 96
11.11 Exam Theory Notes 96

Chapter 12 · SciPy Library 98
12.1 Introduction to SciPy 98
12.2 Need for SciPy 99
12.3 Difference between NumPy and SciPy 100
12.4 Scientific Computing using SciPy 100
12.5 Linear Algebra Basics 101
12.6 Optimization Basics 102
12.7 Statistics using SciPy 103
12.8 Signal Processing Basics 104
12.9 Integration and Interpolation 105
12.10 Applications of SciPy 105
12.11 Exam Theory Notes 106

Chapter 13 · Comparing the Libraries and Worked Programs 108
13.1 NumPy versus Pandas 108
13.2 Matplotlib versus Pandas Visualization 109
13.3 Library Selection Based on Problem Type 110
13.4 Python Libraries Comparison Chart 110
13.5 Worked Program 1 — NumPy Array 111
13.6 Worked Program 2 — Pandas DataFrame 111
13.7 Worked Program 3 — Reading a CSV File 112
13.8 Worked Program 4 — Matplotlib Graph 112
13.9 Worked Program 5 — SciPy Calculation 113
13.10 Mini Data Analysis Project 114
13.11 Python Libraries — Terms and Functions 115
13.12 Exam Theory Notes 115

Conditionals and Iteration 117

Chapter 14 · Boolean Values 118
14.1 Meaning of Boolean Values 118
14.2 True and False in Python 118
14.3 Boolean Data Type 119
14.4 Boolean Expressions 120
14.5 Truthy and Falsy Values 120
14.6 Need for Boolean Logic in Programs 121
14.7 Exam Theory Notes 121

Chapter 15 · Comparison Operators 122
15.1 Meaning of Comparison Operators 122
15.2 Equal To Operator == 123
15.3 Not Equal To Operator != 123

15.4 Greater Than Operator > 124
15.5 Less Than Operator < 124
15.6 Greater Than or Equal To >= 125
15.7 Less Than or Equal To <= 125
15.8 Comparison Operator Examples 126
15.9 Common Mistake: = versus == 126
15.10 Exam Theory Notes 127

Chapter 16 · Logical Operators 128
16.1 Meaning of Logical Operators 128
16.2 and Operator 128
16.3 or Operator 129
16.4 not Operator 129
16.5 Logical Operator Truth Table 130
16.6 Combining Multiple Conditions 130
16.7 Logical Operator Precedence 131
16.8 Real-Life Examples of Logical Conditions 132
16.9 Exam Theory Notes 132

Chapter 17 · Conditional Statements 133
17.1 Meaning of Conditional Statement 133
17.2 Need for Decision Making in Programming 134
17.3 if Statement 134
17.4 if…else Statement 134
17.5 if…elif…else Statement 135
17.6 Nested if Statement 136
17.7 Multiple Conditions in if Statement 136
17.8 Indentation Rules in Conditional Statements 137
17.9 Common Errors in Conditional Statements 138
17.10 Exam Theory Notes 138

Chapter 18 · Conditional Programs 139
18.1 Check Positive, Negative or Zero 139
18.2 Check Odd or Even Number 140
18.3 Find Biggest of Two Numbers 140
18.4 Find Biggest of Three Numbers 140
18.5 Grade Calculation Program 141
18.6 Simple Login Validation Program 142
18.7 Leap Year Checking Program 142
18.8 Electricity Bill Calculation Example 143
18.9 Exam Theory Notes 143

Chapter 19 · Introduction to Iteration 145
19.1 Meaning of Iteration 145
19.2 Need for Loops in Programming 145
19.3 Difference between Conditional and Looping Statements 146
19.4 Entry-Controlled Loop 146
19.5 Loop Components: Initialization, Condition, Update 147
19.6 Infinite Loop Meaning 147
19.7 Loop Execution Flow 148
19.8 Exam Theory Notes 148

Chapter 20 · The while Loop 150
20.1 Meaning of while Loop 150
20.2 Syntax of while Loop 151

20.3 Flow of while Loop 151
20.4 Printing Numbers Using while Loop 152
20.5 Sum of Numbers Using while Loop 152
20.6 Factorial Program Using while Loop 152
20.7 Reverse Number Using while Loop 153
20.8 Common Mistakes in while Loop 153
20.9 Infinite while Loop 154
20.10 Exam Theory Notes 154

Chapter 21 · The for Loop 155
21.1 Meaning of for Loop 155
21.2 Syntax of for Loop 156
21.3 Iterating Over a String 156
21.4 Iterating Over a List 156
21.5 Iterating Over a Tuple 157
21.6 Iterating Over a Range 157
21.7 Difference between for and while Loop 157
21.8 Loop Variable Meaning 158
21.9 Common Mistakes in for Loop 158
21.10 Exam Theory Notes 159

Chapter 22 · The range() Function 160
22.1 Meaning of range() Function 160
22.2 range(stop) 160
22.3 range(start, stop) 161
22.4 range(start, stop, step) 161
22.5 Positive Step Value 162
22.6 Negative Step Value 162
22.7 Printing Number Patterns Using range() 163
22.8 Important Points About range() 163
22.9 Exam Theory Notes 163

Chapter 23 · Loop Control: break, continue and pass 165
23.1 Meaning of Loop Control Statement 165
23.2 Need for Loop Control Statements 165
23.3 break Statement 165
23.4 continue Statement 166
23.5 Difference between break and continue 166
23.6 pass Statement 167
23.7 Loop Control Flow Diagram 167
23.8 Common Mistakes in Loop Control 168
23.9 Exam Theory Notes 168

Chapter 24 · Nested Loops 170
24.1 Meaning of Nested Loop 170
24.2 Nested for Loop 170
24.3 Nested while Loop 171
24.4 Outer Loop and Inner Loop 171
24.5 Multiplication Table Using Nested Loop 172
24.6 Pattern Printing Using Nested Loop 172
24.7 Common Errors in Nested Loops 173
24.8 Exam Theory Notes 173

Chapter 25 · Loop Programs 174
25.1 Print First n Natural Numbers 174

25.2 Sum of First n Numbers 175
25.3 Multiplication Table Program 175
25.4 Factorial of a Number 175
25.5 Count Digits in a Number 176
25.6 Reverse a Number 176
25.7 Palindrome Number Checking 176
25.8 Prime Number Checking 177
25.9 Fibonacci Series Program 177
25.10 Pattern Printing Examples 178
25.11 Exam Theory Notes 178

Chapter 26 · Conditionals and Loops Together 179
26.1 Using if Inside a Loop 179
26.2 Using a Loop Inside an if Statement 180
26.3 Filtering Even Numbers from a List 180
26.4 Counting Positive and Negative Numbers 181
26.5 Menu-Driven Program Using Loop 181
26.6 Password Retry Program 182
26.7 Simple ATM Menu Program 183
26.8 Key Patterns and Common Mistakes 184
26.9 Exam Theory Notes 184

Chapter 27 · Debugging Conditions and Loops 185
27.1 Common Syntax Errors 185
27.2 Indentation Errors 186
27.3 Infinite Loop Errors 186
27.4 Logical Errors in Conditions 187
27.5 Off-by-One Error in Loops 188
27.6 Dry Run Method 188
27.7 Trace Table Method 189
27.8 Exam Theory Notes 189

Chapter 28 · Conditionals and Iteration — Summary 190
28.1 Complete Conditionals Summary 190
28.2 Complete Looping Summary 191
28.3 Important Conditional Terms 191
28.4 Important Looping Terms 191
28.5 Conditional Statement Comparison Chart 192
28.6 Loop Comparison Chart 192
28.7 break, continue and pass — Revision 192
28.8 Exam Theory Notes 193

Functions 195

Chapter 29 · Introduction to Functions 196
29.1 Meaning of Function 196
29.2 Need for Functions 197
29.3 Advantages of Functions 197
29.4 Function as a Reusable Block of Code 197
29.5 Built-in Functions versus User-defined Functions 198
29.6 Exam Theory Notes 198

Chapter 30 · Types of Functions 199
30.1 Built-in Functions 200
30.2 User-defined Functions 200

30.3 Library Functions 200
30.4 Anonymous / Lambda Functions 201
30.5 Recursive Functions 201
30.6 Exam Theory Notes 202

Chapter 31 · Defining and Calling Functions 203
31.1 Function Definition 203
31.2 Function Header 203
31.3 Function Body 204
31.4 Function Call 204
31.5 Flow of Control in Function Execution 205
31.6 Exam Theory Notes 205

Chapter 32 · Parameters and Arguments 206
32.1 Meaning of Parameter 206
32.2 Meaning of Argument 207
32.3 Positional Arguments 207
32.4 Keyword Arguments 207
32.5 Default Arguments 208
32.6 Variable-length Arguments 209
32.7 Parameter versus Argument 209
32.8 Exam Theory Notes 210

Chapter 33 · Return Values 211
33.1 Meaning of Return Statement 211
33.2 Function with Return Value 212
33.3 Function without Return Value 212
33.4 Returning Multiple Values 212
33.5 Difference between return and print() 213
33.6 Exam Theory Notes 213

Chapter 34 · Scope: Local and Global Variables 215
34.1 Meaning of Scope 215
34.2 Local Variable 216
34.3 Global Variable 216
34.4 Local Scope 216
34.5 Global Scope 217
34.6 global Keyword 217
34.7 Lifetime of Variables 218
34.8 Exam Theory Notes 219

Chapter 35 · Function Composition 220
35.1 Meaning of Function Composition 220
35.2 Calling One Function inside Another Function 220
35.3 Nested Function Calls 221
35.4 Step-by-Step Function Composition Example 221
35.5 Benefits of Function Composition 222
35.6 Exam Theory Notes 222

Chapter 36 · Recursion 223
36.1 Meaning of Recursion 223
36.2 Recursive Function 224
36.3 Base Case 224
36.4 Recursive Case 225
36.5 Working of Recursive Calls 225

36.6 Factorial using Recursion 226
36.7 Fibonacci using Recursion 227
36.8 Recursion versus Iteration 227
36.9 Exam Theory Notes 228

Chapter 37 · Function Programs 229
37.1 Simple Function Example 229
37.2 Function with Parameters Example 230
37.3 Function with Return Value Example 230
37.4 Function using Default Argument Example 230
37.5 Function Composition Example 231
37.6 Recursive Function Example 231
37.7 Exam Theory Notes 231

Chapter 38 · Debugging Functions 233
38.1 Common Function Syntax Errors 233
38.2 Missing Function Call Error 233
38.3 Parameter Mismatch Error 234
38.4 Return Value Mistakes 234
38.5 Scope-related Errors 235
38.6 Infinite Recursion Error 235
38.7 Exam Theory Notes 236

Chapter 39 · Functions — Summary 237
39.1 Complete Functions Summary 237
39.2 Important Python Function Terms 238
39.3 Function Types Comparison Chart 238
39.4 Exam Theory Notes 239

Strings 240

Chapter 40 · Introduction to Strings 241
40.1 Meaning of String 241
40.2 Need for Strings in Programming 241
40.3 Creating Strings 242
40.4 Single, Double and Triple Quotes 242
40.5 Empty String 242
40.6 String as a Sequence 243
40.7 Exam Theory Notes 243

Chapter 41 · Accessing Characters 244
41.1 Character Indexing 244
41.2 Positive Indexing 245
41.3 Negative Indexing 245
41.4 Accessing First and Last Character 246
41.5 Index Error in Strings 246
41.6 Exam Theory Notes 246

Chapter 42 · String Slicing 247
42.1 Meaning of Slicing 247
42.2 Basic Slice Syntax 248
42.3 Start and Stop Values 248
42.4 Step Value in Slicing 249
42.5 Reverse String using Slicing 249
42.6 Common Slicing Examples 250

42.7 Slicing Mistakes 250
42.8 Exam Theory Notes 251

Chapter 43 · Immutability of Strings 252
43.1 Meaning of Immutability 252
43.2 Why Strings are Immutable 252
43.3 Modifying a String Indirectly 253
43.4 Reassigning String Variables 253
43.5 Difference between Mutable and Immutable Objects 253
43.6 Exam Theory Notes 254

Chapter 44 · String Operations 255
44.1 String Concatenation 255
44.2 String Repetition 255
44.3 Membership Operators in Strings 256
44.4 Comparison of Strings 256
44.5 Iterating through a String 257
44.6 String Length using len() 257
44.7 Exam Theory Notes 258

Chapter 45 · String Methods 259
45.1 Meaning of String Method 259
45.2 Case Conversion Methods 260
45.3 Search Methods 260
45.4 Testing Methods 261
45.5 Strip Methods 261
45.6 Replace Method 261
45.7 Split and Join Methods 262
45.8 Count Method 262
45.9 startswith and endswith Methods 263
45.10 Important String Methods Table 263
45.11 Exam Theory Notes 263

Chapter 46 · String Formatting 265
46.1 Need for String Formatting 265
46.2 Using format() Method 265
46.3 Using f-Strings 266
46.4 Formatting Numbers inside Strings 266
46.5 Alignment and Spacing 267
46.6 String Formatting Examples 268
46.7 Exam Theory Notes 268

Chapter 47 · The string Module 270
47.1 Meaning of String Module 270
47.2 Importing the String Module 270
47.3 String Constants 271
47.4 ascii_letters, digits and punctuation 271
47.5 Using the String Module in Programs 272
47.6 String Module Worked Examples 272
47.7 Exam Theory Notes 272

Chapter 48 · Lists as Arrays 274
48.1 Meaning of Array 274
48.2 Why Python Uses Lists as Arrays 274
48.3 Creating Lists 275

48.4 Accessing List Elements 275
48.5 Updating List Elements 275
48.6 Lists are Mutable 276
48.7 Difference between Strings and Lists 276
48.8 List Traversal 276
48.9 List Methods 277
48.10 Lists as Character Arrays 277
48.11 Exam Theory Notes 277

Chapter 49 · Converting between Strings and Lists 279
49.1 Converting String to List 279
49.2 Converting List to String 279
49.3 Using split() 280
49.4 Using join() 281
49.5 Character List Example 281
49.6 Word List Example 281
49.7 Exam Theory Notes 282

Chapter 50 · Classic String Programs 283
50.1 Count Vowels and Consonants 283
50.2 Check Palindrome String 283
50.3 Reverse a String 284
50.4 Count Words in a Sentence 284
50.5 Remove Spaces from String 285
50.6 Find Frequency of Characters 285
50.7 Replace Characters in String 285
50.8 Convert Sentence to Title Case 286
50.9 Exam Theory Notes 286

Chapter 51 · Common String Errors 287
51.1 Common String Syntax Errors 287
51.2 Index Out of Range Error 288
51.3 Immutability Error 288
51.4 Wrong Slicing Output 288
51.5 Method Name Mistakes 289
51.6 String and Number Concatenation Error 289
51.7 Exam Theory Notes 289

Chapter 52 · Worked String Examples 290
52.1 Simple String Example 290
52.2 String Indexing Example 291
52.3 String Slicing Example 291
52.4 String Method Example 291
52.5 String Formatting Example 292
52.6 String Module Example 292
52.7 List as Array Example 292
52.8 Complete Mini String Program 293
52.9 Exam Theory Notes 293

Chapter 53 · Strings — Revision 294
53.1 Complete Strings Summary 294
53.2 Important String Terms 295
53.3 String Function and Method Revision Table 295
53.4 String versus List Comparison Chart 296
53.5 Important Exam Points 296

53.6 Exam Theory Notes 296

Lists 298

Chapter 54 · Introduction to Lists 299
54.1 Meaning of List 299
54.2 Need for Lists 300
54.3 Features of Lists 300
54.4 Variable versus List 300
54.5 Real-Life Examples of Lists 300
54.6 Creating Lists 301
54.7 Exam Theory Notes 301

Chapter 55 · Indexing Lists 302
55.1 Indexing in Lists 302
55.2 Positive Indexing 303
55.3 Negative Indexing 303
55.4 Accessing Nested List Elements 303
55.5 Index Out of Range Error 304
55.6 Exam Theory Notes 304

Chapter 56 · List Slicing 305
56.1 Meaning of Slicing 305
56.2 Basic List Slice 305
56.3 Slice with Start and Stop 306
56.4 Slice with Step Value 306
56.5 Reverse List using Slicing 306
56.6 Common Slicing Mistakes 307
56.7 Exam Theory Notes 307

Chapter 57 · List Operations 308
57.1 Concatenation of Lists 308
57.2 Repetition of Lists 308
57.3 Membership Operators in Lists 309
57.4 Length, Minimum, Maximum and Sum 309
57.5 Exam Theory Notes 309

Chapter 58 · Changing Lists 311
58.1 Changing a List Element 311
58.2 Updating Multiple Elements 311
58.3 Adding New Elements 312
58.4 Removing Elements 312
58.5 Replacing List Slices 312
58.6 Exam Theory Notes 313

Chapter 59 · List Methods 314
59.1 append() Method 314
59.2 extend() Method 314
59.3 insert() Method 315
59.4 remove() Method 315
59.5 pop() Method 315
59.6 clear() Method 316
59.7 index() Method 316
59.8 count() Method 316
59.9 sort() Method 317

59.10 reverse() Method 317
59.11 copy() Method 317
59.12 Exam Theory Notes 318

Chapter 60 · List Loops 319
60.1 Traversing using for Loop 319
60.2 Traversing using while Loop 320
60.3 Traversing with Index and range() 320
60.4 Traversing using enumerate() 320
60.5 Traversing Two Lists with zip() 321
60.6 Exam Theory Notes 321

Chapter 61 · Searching and Filtering Lists 322
61.1 Searching an Element 322
61.2 Counting Specific Elements 323
61.3 Filtering List Values 323
61.4 Finding Even and Odd Numbers 323
61.5 Finding Largest and Smallest Elements 324
61.6 Exam Theory Notes 324

Chapter 62 · Mutability 325
62.1 Meaning of Mutability 325
62.2 Changing List Values 326
62.3 List versus String Mutability 326
62.4 Effects of Mutability 326
62.5 Common Mutability Mistakes 327
62.6 Exam Theory Notes 327

Chapter 63 · Aliasing 328
63.1 Meaning of Aliasing 328
63.2 Same List with Two Names 329
63.3 Effect of Changing Aliased Lists 329
63.4 Aliasing versus Copying 330
63.5 Avoiding Aliasing Problems 330
63.6 Exam Theory Notes 330

Chapter 64 · Cloning Lists 331
64.1 Meaning of Cloning 331
64.2 Three Ways to Clone 332
64.3 Shallow Copy and Nested Lists 332
64.4 Exam Theory Notes 333

Chapter 65 · List Parameters 334
65.1 Passing List to Function 334
65.2 Modifying List inside Function 335
65.3 Returning List from Function 335
65.4 Mutable Default Parameter Effect 336
65.5 Safe List Handling in Functions 336
65.6 Exam Theory Notes 336

Chapter 66 · Nested Lists and Matrices 338
66.1 Meaning of Nested List 338
66.2 Accessing Nested List Values 338
66.3 Updating Nested List Values 339
66.4 List as Matrix 339
66.5 Matrix Traversal using Loops 339

66.6 Exam Theory Notes 340

Chapter 67 · Worked List Programs 341

67.1 Simple List Program 341
67.2 Sum and Average of List Elements 341
67.3 Largest and Smallest Number Program 342
67.4 Count Even and Odd Numbers 342
67.5 Search Element in List 342
67.6 Remove Duplicate Elements 343
67.7 Sorting List Example 343
67.8 Nested List Example 344
67.9 Exam Theory Notes 344

Chapter 68 · Common List Errors 345

68.1 Common List Syntax Errors 345
68.2 Index Out of Range Error 346
68.3 Wrong Slicing Error 346
68.4 Mutability Confusion 346
68.5 Aliasing Error 347
68.6 Method Usage Mistakes 347
68.7 Exam Theory Notes 348

Chapter 69 · Lists — Revision 349

69.1 Complete Lists Summary 349
69.2 Important List Terms 350
69.3 List Methods Revision Table 350
69.4 List versus String Comparison Chart 350
69.5 List versus Tuple Comparison Chart 350
69.6 Important Exam Points 351
69.7 Exam Theory Notes 351

Tuples, Dictionaries, Iterators and Generators 352

Chapter 70 · Introduction to Tuples 353

70.1 Meaning of Tuple 353
70.2 Need for Tuple 353
70.3 Tuple Syntax 353
70.4 Creating Empty and Single-Element Tuples 354
70.5 Tuple versus List 354
70.6 Exam Theory Notes 354

Chapter 71 · Accessing Tuples 356

71.1 Positive and Negative Indexing 356
71.2 Tuple Slicing 356
71.3 Nested Tuple Access 357
71.4 Tuple Traversal using Loop 357
71.5 Exam Theory Notes 357

Chapter 72 · Tuple Immutability 358

72.1 Meaning of Immutable Object 358
72.2 Why Tuple Cannot be Modified 359
72.3 Tuple Reassignment 359
72.4 Mutable Objects inside Tuple 360
72.5 Common Tuple Immutability Errors 360
72.6 Exam Theory Notes 360

Chapter 73 · Tuple Operations 362
73.1 Concatenation and Repetition 362
73.2 Membership Operators 362
73.3 Tuple Comparison 362
73.4 Tuple Packing 363
73.5 Tuple Unpacking 363
73.6 Exam Theory Notes 364

Chapter 74 · Tuple Methods and Functions 365
74.1 count() Method 365
74.2 index() Method 365
74.3 Built-in Functions on Tuples 366
74.4 Exam Theory Notes 366

Chapter 75 · Tuple Assignment 367
75.1 Meaning of Tuple Assignment 367
75.2 Multiple Assignment 368
75.3 Swapping Values using Tuple Assignment 368
75.4 Unpacking Tuple Values 368
75.5 Extended Unpacking using * 369
75.6 Common Tuple Assignment Errors 369
75.7 Exam Theory Notes 369

Chapter 76 · Tuples as Return Values 371
76.1 Returning Multiple Values from Function 371
76.2 Receiving Returned Tuple Values 371
76.3 Tuple Return versus List Return 372
76.4 Practical Examples of Tuple Return 372
76.5 Advantages of Tuple Return Values 372
76.6 Exam Theory Notes 372

Chapter 77 · Converting between Lists and Tuples 373
77.1 Converting List to Tuple 373
77.2 Converting Tuple to List 374
77.3 Need for Conversion 374
77.4 Sorting Tuple Data 374
77.5 Exam Theory Notes 374

Chapter 78 · Dictionaries 375
78.1 Meaning of Dictionary 375
78.2 Need for Dictionary 376
78.3 Key-Value Pair Concept 376
78.4 Creating Dictionary 377
78.5 Accessing Dictionary Values 377
78.6 Dictionary versus List versus Tuple 378
78.7 Exam Theory Notes 378

Chapter 79 · Dictionary Operations and Methods 379
79.1 Adding and Updating Items 379
79.2 Deleting Dictionary Items 380
79.3 keys(), values() and items() Methods 380
79.4 get() Method 380
79.5 update() and setdefault() Methods 381
79.6 Dictionary Traversal using Loop 381
79.7 Exam Theory Notes 382

Chapter 80 · Advanced List Processing 383
80.1 List Comprehension 383
80.2 Nested List Comprehension 384
80.3 Filtering and Mapping List Data 385
80.4 Sorting with Key Function 385
80.5 Lambda with List Processing 385
80.6 map(), filter() and reduce() Basics 386
80.7 Exam Theory Notes 386

Chapter 81 · Iterators 387
81.1 Meaning of Iterator 387
81.2 Iterable versus Iterator 388
81.3 iter() and next() Functions 388
81.4 How the for Loop Uses an Iterator 389
81.5 Creating Custom Iterator 389
81.6 Iterator Practical Examples 390
81.7 Exam Theory Notes 390

Chapter 82 · Generators 391
82.1 Meaning of Generator 391
82.2 Need for Generator 392
82.3 yield Keyword 392
82.4 Generator Function 393
82.5 Generator Expression 393
82.6 Generator versus Iterator 393
82.7 Memory Advantage of Generator 394
82.8 Exam Theory Notes 394

Chapter 83 · Worked Examples 395
83.1 Simple Tuple Program 395
83.2 Tuple Assignment Example 396
83.3 Tuple as Return Value Example 396
83.4 Dictionary Count Example 396
83.5 Dictionary Traversal Example 397
83.6 List Comprehension Example 397
83.7 Iterator Example 397
83.8 Generator Example 398
83.9 Complete Mini Program 398
83.10 Exam Theory Notes 399

Chapter 84 · Common Errors 400
84.1 Common Tuple Syntax Errors 400
84.2 Tuple Immutability Error 400
84.3 Unpacking Error 401
84.4 Dictionary Key Error 401
84.5 Wrong Dictionary Method Usage 401
84.6 Iterator StopIteration Error 402
84.7 Generator Logic Mistakes 402
84.8 Exam Theory Notes 403

Chapter 85 · Revision of Part VII 404
85.1 Complete Tuples Summary 404
85.2 Complete Dictionaries Summary 404
85.3 Important Tuple and Dictionary Terms 405

85.4 Tuple versus List Comparison Chart 405
85.5 Dictionary Methods Revision 405
85.6 Iterator versus Generator Comparison Chart 405
85.7 Important Exam Points 406
85.8 Exam Theory Notes 406

TRB Exam Quick Reference 407

Appendix A · Keywords, Built-ins and Operators 408
A.1 The 35 Keywords of Python 3.13 408
A.2 Built-in Functions 409
A.3 Operator Precedence and Associativity 411
A.4 Number Systems and Bitwise Operators 412
A.5 Escape Sequences 413
A.6 Format Specifiers 413

Appendix B · Methods of str, list, tuple, dict and set 414
B.1 How Many Methods? 414
B.2 String Methods 414
B.3 List Methods 416
B.4 Tuple Methods 416
B.5 Dictionary Methods and Operators 417
B.6 Set Methods and Operators 417

Appendix C · Data Types, Conversions and Exceptions 418
C.1 Property Matrix of Built-in Types 418
C.2 Type Conversion Rules 419
C.3 Truth Values 420
C.4 The Exception Hierarchy 421
C.5 try – except – else – finally 422

Appendix D · NumPy, Pandas, Matplotlib and SciPy Cheat Sheets 423
D.1 The Libraries at a Glance 423
D.2 NumPy 423
D.3 Pandas 425
D.4 Matplotlib (pyplot) 426
D.5 SciPy 427

Appendix E · Commonly Confused Pairs 428
E.1 Look-alike Operators, Methods and Ideas 428

Appendix F · Predict-the-Output Bank 431
F.1 Operators, Numbers and Assignment 431
F.2 Strings 432
F.3 Lists 433
F.4 Tuples, Dictionaries and Sets 433
F.5 Functions, Arguments and Scope 434
F.6 Conditionals and Loops 435
F.7 Iterators, Generators and Functional Tools 436

Appendix G · History, Standards, Tools and Glossary 436
G.1 Python Timeline 436
G.2 PEPs and the Zen of Python 437
G.3 Implementations of Python 438
G.4 IDEs, Distributions and Package Tools 438
G.5 Glossary 438

Appendix H · Last-Day Revision 440
H.1 Python Basics 440
H.2 Python Libraries 440
H.3 Conditionals and Iteration 441
H.4 Functions 441
H.5 Strings 441
H.6 Lists 442
H.7 Tuples, Dictionaries, Iterators and Generators 442

Appendix I · Standard Library Modules 443
I.1 Importing Modules 443
I.2 math 443
I.3 random and statistics 444
I.4 string 445
I.5 collections, itertools, functools and copy 445
I.6 sys, os, time and datetime 446

Appendix J · Syntax of Every Statement 447
J.1 Statement Templates 447
J.2 File Modes 449

PART I · SYLLABUS 1

Python Basics

Chapter 1 Introduction to Python

Chapter 2 The Python Interpreter

Chapter 3 Interactive Mode and Script Mode

Chapter 4 Errors and Debugging

Chapter 5 Values and Data Types

Chapter 6 Variables in Python

Chapter 7 Expressions, Statements, Precedence and Comments

Python interpreter and interactive mode · debugging · data types (int, float, Boolean, string, list) · variables · expressions · statements · tuple assignment · precedence of operators · comments.

CHAPTER · UNIT 1 BASICS

1 Introduction to Python

OPENING SITUATION

In the college computer laboratory a student types one line, `print("Hello, Python!")`, presses Enter, and the message appears at once. No project set-up, no declarations, no separate build step. This chapter explains what Python is, why it became so popular, where it is used, and what really happens between typing that line and seeing the output.

IN THIS CHAPTER

- 1.1 Meaning of Python
- 1.2 Features of Python
- 1.3 Applications of Python
- 1.4 Python as an Interpreted Language
- 1.5 Python Program Execution Process
- 1.6 Exam Theory Notes

YOU NEED

- Everyday use of a computer: files, folders, typing text.
- No programming experience is assumed.

1.1 Meaning of Python

DEFINITION 1.1 — PYTHON

Python is a high-level, general-purpose programming language. We use it to write instructions (code) that a computer carries out to perform a task. Its design puts **readability** and **simplicity** first, so it is easy to learn and pleasant to use.

2 Main idea

- Clean, readable syntax that looks close to English.
- Easy for beginners, yet powerful enough for professionals.
- Used in many fields because it saves time and effort.

3 Why the name "Python"?

The creator, **Guido van Rossum**, named it after the British comedy group *Monty Python*, whose show he enjoyed. The name has nothing to do with the snake.

4 Simple explanation

Python lets people write instructions in a form that is easy to read. We write the code; the Python system translates and runs it so that the computer does the job.

5 Key characteristics

- **High-level** — close to human language; memory is managed automatically.
- **Interpreted** — run by the Python interpreter (see 1.4).
- **Object-oriented** — supports classes and objects.
- **Open source** and **portable** — Windows, macOS, Linux and more.

The smallest complete Python program is one line. The editor window on the left is the program file; the dark screen on the right is exactly what the computer prints when the file is run.

hello.py

```
1 print("Hello, Python!")
```

Output

```
Hello, Python!
```

Real-life uses

E Education
Teaching programming, learning tools, online courses.

W Web development
Websites and web applications (Django, Flask).

A Automation
Scripts that do repetitive tasks and save time.

D Data science
Analysing data, drawing charts, making predictions.

AI Artificial intelligence
Smart systems, chatbots, machine-learning models.

! Exam points
Definition · characteristics · origin of the name · uses · a basic example.

★ **Python is an easy, powerful and widely used programming language.**

1.2 Features of Python

Features are the important qualities that make Python a popular and powerful language used worldwide.

1 Core features

- **Easy syntax** — simple and clear, similar to English.
- **Readability** — indentation shows structure, so code is easy to follow.
- **Simplicity** — less code for more work.
- **Beginner-friendly** — ideal for students.

2 Portable and open source

- The same program runs on Windows, Linux and macOS.
- Free to use, modify and distribute.
- A large global community develops and supports it.

3 Dynamic typing

- The type belongs to the *value*, and is checked while the program runs.
- No type declaration is needed.
- One variable may refer to values of different types at different times.

4 Powerful support

- **Object-oriented** — classes, objects, inheritance, polymorphism.
- **Large standard library** — "batteries included".
- **Third-party packages** — a huge ecosystem on PyPI (the Python Package Index).

dynamic.py

```
1  x = 10              # x refers to an integer
2  print(x, type(x))
3  x = "Python"        # now x refers to a string
4  print(x, type(x))
```

Output

```
10 <class 'int'>
Python <class 'str'>
```

Screen 1.1 Dynamic typing: the same name `x` first refers to an `int` and then to a `str`; `type()` reports the type of the current value.

✓ Faster development
Write less code and get more done quickly.

✓ Versatile
Web, data science, AI, machine learning, automation.

✓ Reliable
Stable and trusted by industry worldwide.

EXAM POINT

List any five or six features · give an example of dynamic typing · interpreted versus compiled · why Python is popular · meaning of open source.

★ **Python is simple, powerful, portable and rich in features.**

1.3 Applications of Python

Applications of Python are the real-world areas and industries where it is used to solve problems, build systems and create tools.



Figure 1.1 Main application areas of Python, with typical libraries and frameworks used in each.

Table 1.1 Other uses of Python

Area	What Python does there	Typical tools
Desktop applications	Cross-platform programs with windows and buttons	Tkinter, PyQt, Kivy
Game development	Simple 2D games	Pygame
Networking	Network tools, sockets, web scrapers	`socket`, `requests`
Cybersecurity	Security testing, scanning, automation	Scapy and similar
Internet of Things (IoT)	Programming microcontrollers and connected devices	MicroPython

snippets.py

```
print("Automation")      # a one-line script
import pandas as pd      # the usual way to load pandas for data analysis
```

★ **Python is used in education, web, AI, data science and automation.**

1.4 Python as an Interpreted Language

DEFINITION 1.2 — INTERPRETED LANGUAGE

Python is called an **interpreted language** because the program is run by an **interpreter**: a program that reads the source code and carries it out directly, so the result appears immediately. We never produce a separate machine-code `.exe` file before running.

How it works

1. We write the program in Python (the **source code**, a `.py` file).
2. The interpreter reads the code and checks its syntax.
3. It **compiles** the whole file into **bytecode** — a compact, platform-independent intermediate form.
4. The **Python Virtual Machine (PVM)** executes the bytecode instructions one by one and produces the output.

COMMON MISTAKE 1.1 — "PYTHON RUNS LINE BY LINE WITH NO COMPILATION"

In the standard interpreter (CPython) the whole file is first compiled to bytecode, and only then executed. That is why a `SyntaxError` on line 10 stops the program *before* line 1 prints anything. The "line by line" feeling comes from the interactive shell, where each statement you type is compiled and run as soon as you press Enter.

1 Line-by-line idea

In the shell you can run a few lines, check the result and continue — useful for learning and debugging.

2 Role of interpreter

Checks syntax · translates to bytecode · runs it on the PVM · shows output or an error message.

3 Advantages

Instant feedback, interactive shell, faster development and testing.

LIMITATION

Interpreted Python is usually slower than compiled languages such as C/C++ for heavy computation. Libraries like NumPy recover speed by doing the heavy work in compiled code.

>>> Python Shell

```
>>> print("Hello")
Hello
```

Screen 1.2 In the Python shell the line runs the moment Enter is pressed, and the result is shown at once.

★ **Python uses an interpreter to execute programs and give quick results.**

1.5 Python Program Execution Process

The **execution process** is the sequence of steps a Python program goes through from the moment we write it to the moment we get the output (or an error).

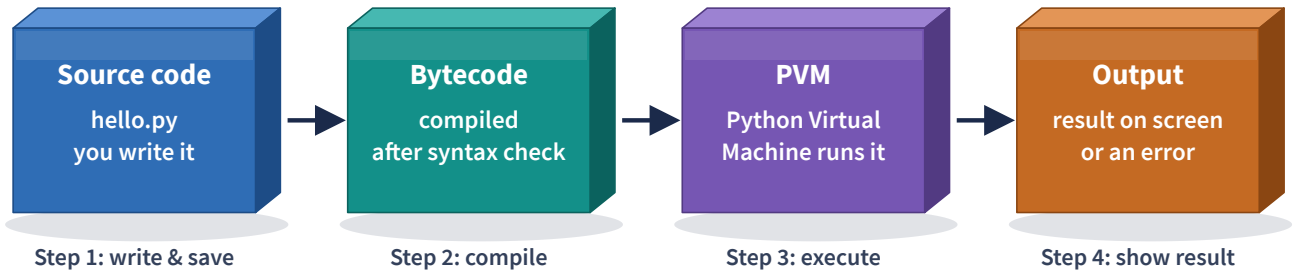


Figure 1.2 The four stages of running a Python program. Syntax errors stop the process at the bytecode stage; logic errors let it run but give wrong output.

1 Write source code

Type the program in a text editor or IDE (Integrated Development Environment) and save it with the `.py` extension, e.g. `hello.py`.

2 Save and run

Run it from a terminal with `python hello.py`, or press the Run button in IDLE, VS Code or PyCharm.

3 Bytecode generation

Python checks the syntax; if correct, it compiles the source into bytecode, a low-level, platform-independent form.

4 PVM execution and output

The PVM loads the bytecode, executes it instruction by instruction, manages memory and shows the output — or an error message.

▶ Terminal

```
C:\lab> python hello.py
Welcome
```

COMMON MISTAKE 1.2 — "EVERY RUN CREATES A .PYC FILE"

Bytecode for the script you run is kept in memory only. A `.pyc` file is saved (in a `__pycache__` folder) for **imported modules**, so that the next import can skip compilation.

★ **A Python program is written, translated to bytecode, executed by the PVM, and then produces output.**

1.6 Exam Theory Notes

TRB EXAM THEORY — KEY FACTS

1. Python was created by **Guido van Rossum** at CWI, Netherlands; work began in December 1989 and the first public release (0.9.0) came in **February 1991**.
2. The name comes from the BBC comedy **Monty Python's Flying Circus**, not from the snake.
3. Python 1.0 — January 1994; Python 2.0 — October 2000 (list comprehensions, garbage collector); Python 3.0 — **December 2008** (not backward compatible).
4. Python 2 reached end of life on **1 January 2020**; Python 2.7 was its last version.
5. Python is **high-level, interpreted, dynamically typed, strongly typed, object-oriented** and general-purpose; it supports procedural, object-oriented and functional styles.
6. **Dynamically typed**: the type belongs to the value and is checked at run time; a name can later refer to a value of another type.
7. **Strongly typed**: no silent mixing of unrelated types — `"5" + 5` is a TypeError.

8. Python is **free and open source**, distributed under the **PSF License** (GPL-compatible); it is managed by the **Python Software Foundation** (founded 2001).
9. Guido was called the **BDFL** (Benevolent Dictator For Life) until 2018; Python is now governed by an elected five-member **Steering Council**.
10. Changes are proposed through **PEPs** (Python Enhancement Proposals). PEP 8 = style guide; PEP 20 = The Zen of Python (`import this`); PEP 257 = docstring conventions.
11. Python is **portable** (platform independent): the same source runs on Windows, Linux and macOS wherever an interpreter exists.
12. Blocks are marked by **indentation**, not braces; PEP 8 recommends **4 spaces** per level.
13. Python is called "batteries included" because of its large **standard library**; extra packages come from **PyPI** and are installed with **pip**.
14. Python source files use the extension **.py**; compiled bytecode files use **.pyc**; `.pyw` runs on Windows without a console window.
15. Python is **case-sensitive**: `Total` , `total` and `TOTAL` are three different names.
16. Main application areas: web development (Django, Flask), data science (NumPy, Pandas), machine learning (scikit-learn, TensorFlow, PyTorch), automation and scripting, GUI (Tkinter), games (Pygame) and education.
17. Limitations: slower than compiled languages such as C, higher memory use, weak in mobile development, run-time type errors.

PREDICT THE OUTPUT — EVERY RESULT VERIFIED BY RUNNING IT

`type(10)`	`<class 'int'>`	`type(10.0)`	`<class 'float'>`
`sys.version_info.major`	`3`	`"5" + 5`	`TypeError`
`Total = 1; total`	`NameError`	`len(keyword.kwlist)`	`35`

Chapter 1 Summary

- Python is a high-level, general-purpose, readable language created by Guido van Rossum.
- Key features: easy syntax, dynamic typing, portability, open source, object-oriented, large library.
- Used in web, data science, AI, automation, education, games, networking and IoT.
- The interpreter compiles source to bytecode; the PVM executes it.
- Syntax errors stop the run before execution; logic errors give wrong output.
- `.pyc` files are cached bytecode of imported modules.

Next — Chapter 2: the Python interpreter and its interactive and script modes.

CHAPTER · UNIT 1 BASICS

2 The Python Interpreter

OPENING SITUATION

A new laboratory computer has just been set up. Before anyone can run a program, Python must be installed, checked and opened. This chapter explains the program that does all the work — the interpreter — how it differs from a compiler, and how to type the first command at the `>>>` prompt.

IN THIS CHAPTER

- 2.1 Meaning of Interpreter
- 2.2 Role of the Python Interpreter
- 2.3 Interpreter versus Compiler
- 2.4 Installing and Opening the Python Interpreter
- 2.5 Writing the First Python Command
- 2.6 Exam Theory Notes

YOU NEED

- Chapter 1: source code, bytecode and the PVM.

2.1 Meaning of Interpreter

DEFINITION 2.1 — INTERPRETER

An **interpreter** is a program that acts as a translator: it reads source code written in a high-level language and executes it step by step. It does **not** create a separate executable file; it runs the program directly.



1 Main idea

- Turns high-level instructions into actions the computer can carry out.
- Executes each instruction immediately and shows quick results.

2 Why it is useful

- No extra build steps.
- Easy to test small ideas.
- Errors are found and fixed quickly.
- Perfect for learning and experimenting.

IN SIMPLE WORDS

An interpreter is like a live translator: you say (type) one instruction, it translates it into a form the computer understands, and the computer performs it straight away.

`>>>` Python Shell

```
>>> 2 + 3
5
```

Screen 2.1 The interpreter reads the expression `2 + 3`, evaluates it and displays the result `5`.

★ **An interpreter reads, translates and executes a program step by step.**

2.2 Role of the Python Interpreter

The **Python interpreter** is the program that runs Python code. It is the bridge between the programmer and the computer, and it makes sure programs are executed correctly. Its job has five parts:

1 Reads Python code

From two sources: the **interactive shell** (REPL — Read, Evaluate, Print, Loop) and **script files** (`.py`).

2 Checks syntax

Looks for syntax errors first; a mistake is reported at once, before anything runs.

3 Translates code

Correct source code is compiled into **bytecode**, a low-level intermediate form that the PVM can execute.

4 Executes instructions

The Python Virtual Machine runs the bytecode: calculations, comparisons, loops, memory handling and other run-time tasks.

5 Displays output or errors

Shows the result, or a suitable error message if something goes wrong. In interactive mode it also shows the `>>>` prompt for the next command.

The syntax check really happens before execution. In the screen below line 1 is correct, but nothing is printed, because the whole file is rejected while it is being compiled.

check.py

```
1 print("Line 1 runs?")
2 print("Line 2"
```

Output

```
  File "check.py", line 2
    print("Line 2"
          ^
SyntaxError: '(' was never closed
```

Screen 2.2 A missing `)` on line 2 is caught by the syntax check, so not even line 1 is executed.

★ **The Python interpreter reads, checks, translates, executes and displays the result.**

2.3 Interpreter versus Compiler

Both an interpreter and a compiler are **language translators**: they convert high-level code written by people into a form the computer can execute. They differ in *when* and *how much* they translate.

I Interpreter

- Translates and executes the program as it runs.
- Stops and shows an error as soon as it reaches a problem.
- No separate executable file.

Flow: source → interpreter → execute

C Compiler

- Reads the entire program at once.
- Translates the whole program into machine code before execution.
- Creates a separate executable file.

Flow: source → compiler → .exe → execute

Table 2.1 Interpreter versus compiler

Point	Interpreter	Compiler
Translation	During execution, statement by statement	Whole program before execution
Speed	Slower (work is done during every run)	Faster (translation done once)
Debugging	Easier — errors shown as they are reached	Harder — errors listed after compilation
Output file	No separate executable file	Creates an executable file
Examples	Python, JavaScript	C, C++

Advantages of an interpreter

- Easy debugging and testing; immediate feedback.
- Ideal for interactive programming.
- No compile-again step after each change.

Advantages of a compiler

- Faster execution and optimised machine code.
- Better for large, performance-critical programs.
- The executable can be shared and run directly.

COMMON MISTAKE 2.1 — "PYTHON NEVER COMPILES"

CPython is a hybrid: it **compiles** the whole source file to bytecode, then **interprets** the bytecode on the PVM. What it does not produce is a native machine-code executable. "Interpreted" describes this execution model, not a total absence of compilation.

★ **An interpreter translates as the program runs; a compiler translates the whole program before it runs.**

2.4 Installing and Opening the Python Interpreter

Installing Python means downloading and setting it up on the computer. **Opening the interpreter** means starting the Python environment where we type commands and see results instantly.



1 Install Python

- Go to the official website **python.org** and download the latest Python 3.x.
- Run the installer and tick **"Add Python to PATH"** before clicking **Install Now**.

2 Verify installation

Open Command Prompt (Windows) or Terminal (macOS/Linux) and type `python --version` (or `py --version` on Windows). The installed version is printed.

3 Open the interpreter

Type `python` (or `py`) and press Enter. The interactive interpreter starts and shows `>>>`.

4 Open IDLE

IDLE (Integrated Development and Learning Environment) is installed with Python. Windows: Start → IDLE (Python 3.x). macOS/Linux: Applications/Menu → IDLE. It opens the Python shell automatically.

▶ Command Prompt

```
C:\lab> python --version
Python 3.13.10

C:\lab> python
Python 3.13.10 (tags/v3.13.10, ...) [MSC v.1944 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
>>> print("Hello!")
Hello!
>>> exit()
```

Screen 2.3 Checking the version and opening the interpreter. The `>>>` prompt means Python is ready for a command; `exit()` (or Ctrl+Z then Enter on Windows, Ctrl+D on Linux/macOS) leaves it.

Table 2.2 Common installation problems

Problem	Cause	Fix
"'python' is not recognized as an internal or external command"	Python is not on the system PATH	Re-run the installer and tick "Add Python to PATH", or add the Python and Scripts folders to PATH
Wrong version starts	An old or second installation is found first	Use the latest stable Python 3.x; on Windows `py -3.13` picks a version
Exam tip	Always verify with `python --version` or `py --version` before starting.	

★ **Install Python, open the interpreter from a terminal or IDLE, and start typing commands at the >>> prompt.**

2.5 Writing the First Python Command

The **first Python command** is the first instruction typed in the interpreter. It is usually used to test that Python is working. Open the shell (IDLE, a terminal or any Python environment), wait for `>>>`, type the command and press Enter — the output appears on the next line.

>>> Python Shell

```
>>> print("Hello, Python!")
Hello, Python!
>>> 10 + 20
30
>>> print(5 * 2)
10
```

Screen 2.4 `print()` displays text or the result of an expression. In the shell, a bare expression such as `10 + 20` is also echoed automatically.

Common mistakes with the first command

>>> Python Shell

```
>>> print(Hello, Python!)
  File "<stdin>", line 1
    print(Hello, Python!)
                       ^
SyntaxError: invalid syntax
>>> print "Hello"
  File "<stdin>", line 1
    print "Hello"
    ^^^^^^^^^^^^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?
>>> pritn("Hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pritn' is not defined. Did you mean: 'print'?
```

Screen 2.5 Real error messages: text without quotes is not valid code; `print` needs parentheses in Python 3; a misspelt name is not defined.

Table 2.3 First-command mistakes and fixes

Mistake	Wrong	Why wrong	Correct
Missing quotes	`print(Hello, Python!)`	Text must be inside quotes	`print("Hello, Python!")`
Missing parentheses	`print "Hello"`	In Python 3 `print` is a function and needs `( )`	`print("Hello")`
Spelling error	`pritn("Hello")`	`pritn` is not a function	`print("Hello")`

★ **The first Python command is usually typed in the Python shell to test that Python works.**

2.6 Exam Theory Notes

TRB EXAM THEORY — KEY FACTS

1. An **interpreter** translates and executes a program statement by statement; a **compiler** translates the whole program into machine code before it runs.
2. CPython first **compiles** source code to platform-independent **bytecode**, which the **Python Virtual Machine (PVM)** then executes — so Python is both compiled and interpreted.
3. Execution pipeline: source (.py) → compiler → bytecode (.pyc) → PVM → output.
4. Bytecode of **imported modules** is cached in the `__pycache__` folder as files such as `mod.cpython-313.pyc`; the main script's bytecode is not saved.
5. The `dis` module disassembles a function into bytecode instructions: `dis.dis(f)`.
6. **Implementations**: **CPython** (reference, written in C), **Jython** (Java/JVM), **IronPython** (.NET), **PyPy** (Python with a JIT compiler — usually fastest), **MicroPython** (microcontrollers), Brython (browser).
7. A **JIT** (just-in-time) compiler translates frequently used bytecode to machine code while the program runs; PyPy uses one. CPython 3.13 added an experimental JIT.
8. Interpreter errors are reported at the line where they occur; execution stops at the first unhandled error, but lines before it have already run.
9. **Syntax errors** are found during compilation, so no line of a file with a syntax error runs at all.
10. Commands: `python` (or `py` on Windows) starts the interpreter; `python file.py` runs a script; `python -m module` runs a library module; `python --version` shows the version.

11. **IDLE** (Integrated Development and Learning Environment) is the IDE bundled with Python, written using Tkinter. Other IDEs: PyCharm, VS Code, Spyder, Jupyter Notebook, Thonny.
12. Python manages memory automatically using **reference counting** plus a cyclic **garbage collector** (module `gc`).
13. `sys.getrefcount(x)` reports the reference count of an object; `sys.version` gives the version string.
14. Compiler versus interpreter: a compiler reports all errors after scanning the whole program and produces an executable; an interpreter stops at the first error and needs the source each time.

PREDICT THE OUTPUT — EVERY RESULT VERIFIED BY RUNNING IT

`compile("1 + 2", "<s>", "eval").co_name`	`'<module>'`	`eval("2 ** 10")`	`1024`
`type(compile("x = 5", "f", "exec")).__name__`	`'code'`	`import dis; type(dis.Bytecode(lambda: 1)).__name__`	`'Bytecode'`

Chapter 2 Summary

- An interpreter reads and executes code directly, without an executable file.
- The Python interpreter reads code from the shell or `.py` files, checks syntax, compiles to bytecode, runs it on the PVM and shows output or errors.
- Compiler: whole program first, executable file, faster runs. Interpreter: easier debugging, instant feedback.
- Install from python.org with "Add Python to PATH"; verify with `python --version`.
- Open with `python` / `py` or IDLE; `>>>` means ready.
- `print()` needs parentheses and quoted text.

Next — Chapter 3: interactive mode and script mode.

CHAPTER · UNIT 1 BASICS

3 Interactive Mode and Script Mode

OPENING SITUATION

During a lab session one student tries quick sums at the `>>>` prompt, while another types a ten-line marks program into a file and runs it. Both are using Python, but in two different modes. This chapter explains both modes, the IDLE environment, and how to write, save and run a program file.

IN THIS CHAPTER

- 3.1 Meaning of Interactive Mode
- 3.2 Using the Python Shell and IDLE
- 3.3 Meaning of Script Mode
- 3.4 Writing and Saving Python Programs
- 3.5 Interactive Mode versus Script Mode
- 3.6 Exam Theory Notes

YOU NEED

- Chapter 2: opening the interpreter.

3.1 Meaning of Interactive Mode

DEFINITION 3.1 — INTERACTIVE MODE

Interactive mode is the way of working in which commands are typed directly into the interpreter and executed immediately. Type one command, press Enter, get the result at once.

1 Type command at the >>> prompt → 2 Interpreter reads compiles it → 3 Executes immediately → 4 Displays result then a new prompt

1 Why useful

Quick testing of commands, trying small expressions, checking output instantly, easy learning.

2 In simple words

A live conversation with Python: you type, it answers at once. `>>>` shows it is ready for the next command.

3 Uses

Quick tests, learning syntax, trying formulas, debugging — checking an output and finding a small mistake.

`>>>` Python Shell

```
>>> 5 + 7
12
>>> print("Hello")
Hello
>>> 15 / 4
3.75
```

Screen 3.1 Each line is executed as soon as Enter is pressed.

★ **Interactive mode lets the user type commands and get results immediately.**

3.2 Using the Python Shell and IDLE

S Python Shell

- The interactive environment where commands are typed and run immediately.
- Shows the prompt `>>>`, accepts one command at a time, gives immediate output.

I IDLE

- **Integrated Development and Learning Environment**.
- Comes with Python; beginner-friendly.
- Has both a **Shell window** and an **editor window** for files.

Install Python → **Open IDLE** Start menu / Applications → **— or — Terminal** type python or py → **Shell shows >>>** ready for input

▶ IDLE Shell 3.13.10

```
Python 3.13.10 (tags/v3.13.10, ...) [MSC v.1944 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>> print("Welcome")
Welcome
>>> 10 * 2
20
>>>
```

Screen 3.2 The IDLE shell window. Uses: practising commands, testing expressions, learning syntax, instant output.

EXAM POINT

Full form of IDLE · meaning of Python shell · prompt symbol `>>>` · a simple example.

★ **The Python shell and IDLE let users run commands interactively and learn Python easily.**

3.3 Meaning of Script Mode

DEFINITION 3.2 — SCRIPT MODE

Script mode is the mode in which Python code is written in a file (a *script*, saved with the `.py` extension) and then executed as a complete program.

1 Write code many lines → **2 Save as .py** example.py → **3 Run using Python** F5 or python example.py → **4 Output appears**

✓ Why useful

- Suitable for long and complete programs.
- Easy to edit, save, reuse and share.
- Best choice for real projects.

≡ In simple words

Like writing a full note in a notebook: write all the instructions line by line, save the file, then ask Python to run the whole file at once.

example.py

```
1 name = "Anu"
2 print("Hello", name)
```

▶ Output

```
Hello Anu
```

Screen 3.3 A two-line script. `print()` separates its items with a space.

★ **Script mode lets Python programs be written in a file, saved and executed as a complete program.**

3.4 Writing and Saving Python Programs

Writing and saving a program means creating Python statements in an editor and storing them in a `.py` file.

1. **Write** — open IDLE (File → New File), a text editor or VS Code and type the statements clearly.
2. **Save** — File → Save, give a name such as `hello.py`; keep the `.py` extension.
3. **Run** — Run → Run Module (F5) in IDLE, or type `python hello.py` in a terminal opened in that folder.

hello.py

```
1 print("My first saved program")
```

▶ Output

```
My first saved program
```

▶ Command Prompt

```
C:\lab> python hello.py
My first saved program
```

Table 3.1 Common mistakes when saving and running

Mistake	Why it happens	Fix
Forgetting `.py`	File saved as plain text	Always save as `name.py`
Not saving the file	Running an old version	Save before running (IDLE asks)
Wrong filename / folder	Name typed wrongly or terminal in another folder	Check the name; `cd` to the file's folder
Syntax error in code	Mistake in a statement	Read the error line, fix, save, run again

TIPS

Use meaningful filenames, avoid spaces and odd symbols, and never name your file after a module you import (for example `numpy.py` or `random.py`) — Python would import your file instead of the real module.

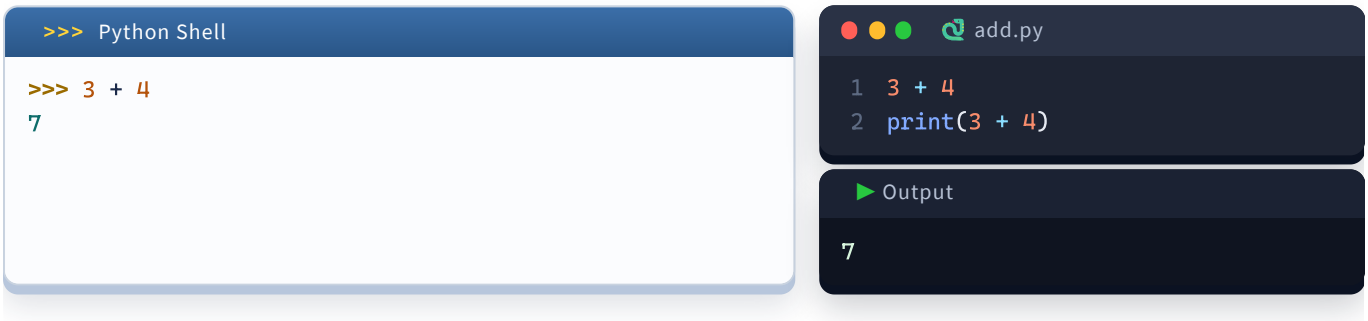
★ **A program is written in an editor, saved as a .py file and run to produce output.**

3.5 Interactive Mode versus Script Mode

Table 3.2 Interactive mode versus script mode

Point	Interactive mode	Script mode
How code is entered	Typed in the Python shell	Written in a file
Execution	One statement at a time	The whole file as one program
Saving	Not needed (lost when the shell closes)	Required (`.py` file)
Output	Immediately after each command; expression values echoed	After running the file; only what `print()` shows
Best use / size	Quick testing, learning; small snippets	Projects; medium to large programs
Reuse	Limited	Easy to reuse and share

A key practical difference: the shell automatically displays the value of an expression, but a script does not. The same line behaves differently in the two modes.



Screen 3.4 In the shell `3 + 4` shows `7`. In a script the bare expression is computed but not shown — only `print()` produces output.

Advantages of interactive mode

Quick feedback, easy to test ideas, simple for beginners.

Advantages of script mode

Better organisation, code reuse and sharing, suitable for real-world projects.

★ **Interactive mode is best for quick testing; script mode is best for complete programs.**

3.6 Exam Theory Notes

TRB EXAM THEORY — KEY FACTS

1. **Interactive mode** (the shell / REPL) shows the prompt `>>>`; each statement runs immediately when Enter is pressed. REPL = Read–Evaluate–Print–Loop.
2. The secondary prompt `...` appears while a compound statement (if, for, def) is being typed; a blank line ends it.
3. In interactive mode, the value of an **expression** is echoed automatically; in script mode, nothing is shown unless `print()` is used.
4. The special variable `_` holds the last echoed result in interactive mode only.
5. **Script mode**: statements are saved in a .py file and executed together; the program can be saved, edited and rerun.
6. In IDLE, a new file is opened with **File → New File** (Ctrl+N), and run with **Run → Run Module (F5)**; the output appears in the shell window.
7. Interactive mode is for testing small pieces of code; script mode is for complete programs.
8. Exit the shell with `exit()`, `quit()`, Ctrl+Z then Enter (Windows) or Ctrl+D (Linux/macOS).

9. `help()` starts the interactive help utility; `help(len)` shows help on one object; `dir()` lists names in the current scope.
10. `if __name__ == "__main__":` — the code beneath runs only when the file is executed directly, not when it is imported; `__name__` is `"__main__"` for the running script.
11. A shebang line `#!/usr/bin/env python3` lets a script run directly on Unix-like systems.
12. `python -i file.py` runs a script and then stays in interactive mode with its variables available.

PREDICT THE OUTPUT — EVERY RESULT VERIFIED BY RUNNING IT

`__name__`	`'__main__'`
`type(help).__name__`	`'_Helper'`
`print("A", "B")`	`A B`

Chapter 3 Summary

- Interactive mode: type at `>>>`, result shown at once.
- IDLE = Integrated Development and Learning Environment: shell + editor.
- Script mode: code saved in a `.py` file and run as a whole.
- Run a script with F5 in IDLE or `python file.py`.
- Only the shell echoes expression values; scripts need `print()`.
- Save before running; keep the `.py` extension.

Next — Chapter 4: errors and debugging.

CHAPTER · UNIT 1 BASICS

4 Errors and Debugging

OPENING SITUATION

A student's average-marks program runs without complaint but prints 22.5 when the answer should be 15.0. Another student's program will not start at all because of a missing colon. Both programs contain errors — of different kinds. This chapter names the three kinds of error, shows how to read Python's error messages, and gives a simple method for finding and fixing bugs.

IN THIS CHAPTER

- 4.1 Meaning of Error
- 4.2 Syntax Errors
- 4.3 Runtime Errors
- 4.4 Logical Errors
- 4.5 Error Messages and Tracebacks
- 4.6 Simple Debugging Techniques
- 4.7 Exam Theory Notes

YOU NEED

- Chapters 2–3: running code in the shell and as a script.

4.1 Meaning of Error

DEFINITION 4.1 — ERROR

An **error** (a *bug*) is a mistake in a program that stops it from running correctly or makes it give wrong output. Errors show that something in the code needs correction; they are a normal part of learning to program.

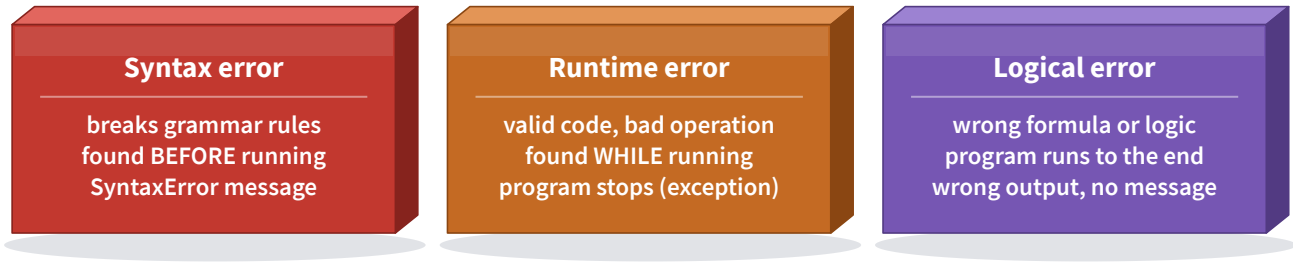


Figure 4.1 The three kinds of error, when each is detected and what the programmer sees.

Why errors happen

- Spelling mistakes in commands; wrong symbols or punctuation.
- Wrong order of steps; invalid or unexpected input.
- Wrong logic or thinking.

Why errors help

Error messages show exactly where the problem is; fixing them improves the program and our understanding of how the computer reacts.

★ An error is a mistake that prevents correct execution or gives wrong output.

4.2 Syntax Errors

DEFINITION 4.2 — SYNTAX ERROR

A **syntax error** is caused by breaking Python's grammar rules. Python checks the syntax of the whole file **before** running it, so the program stops before execution and shows a `SyntaxError` message.

1 Common causes

Missing colon `:` · unmatched brackets `() [] {}` · missing quotes · wrong indentation · misspelt keywords.

2 In simple words

Like a grammar mistake in a sentence: if the sentence is wrong, the reader cannot understand it.

3 Python's reaction

Nothing runs; the message points to the line and position with `^`.

main.py

```
1 x = 8
2 if x > 5
3     print(x)
```

Output

```
  File "main.py", line 2
    if x > 5
           ^
SyntaxError: expected ':'
```

>>> Python Shell

```
>>> print([1, 2, 3)
  File "<stdin>", line 1
    print([1, 2, 3)
                  ^
SyntaxError: closing parenthesis ')' does not match opening parenthesis '['
>>> print("Hello)
  File "<stdin>", line 1
    print("Hello)
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

Screen 4.1 Real syntax errors: a missing colon after `if`, a mismatched bracket, an unterminated string.

Table 4.1 Syntax mistakes and corrections

Mistake	Wrong code	Correct code	Why it is wrong
Missing colon	`if x > 5`	`if x > 5:`	A colon is required after `if`, `for`, `while`, `def`, `class` ...
Missing bracket	`print([1, 2, 3)`	`print([1, 2, 3])`	Opening and closing brackets must match
Wrong quote	`print("Hello)`	`print("Hello")`	A string must start and end with the same quote

★ **A syntax error happens when code does not follow Python's grammar rules.**

4.3 Runtime Errors

DEFINITION 4.3 — RUNTIME ERROR (EXCEPTION)

A **runtime error** occurs while the program is running. The syntax is correct and the program starts, but it stops ("crashes") when it reaches an instruction that cannot be carried out. Python calls such an error an **exception**.

Table 4.2 Common runtime errors

Cause	Example	Exception
Division by zero	`10 / 0`	`ZeroDivisionError`
Invalid type conversion	`int("abc")`	`ValueError`
Wrong type in an operation	`"5" + 2`	`TypeError`
Index out of range	`[1, 2, 3][5]`	`IndexError`
Undefined name	`print(total)`	`NameError`
File not found	`open("x.txt")`	`FileNotFoundError`

main.py

```
1 print("Start")
2 print(10 / 2)
3 print(10 / 0)
4 print("End")
```

Output

```
Start
5.0
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(10 / 0)
          ~~~^~~
ZeroDivisionError: division by zero
```

Screen 4.2 Lines 1 and 2 run; line 3 raises `ZeroDivisionError`, so line 4 is never reached.

>>> Python Shell

```
>>> int('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
>>> "5" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

FIXING A RUNTIME ERROR

1 Identify the line number in the traceback · 2 read the error name (`ZeroDivisionError` , `ValueError` , `FileNotFoundError` ...) · 3 check the input data and values · 4 correct the operation and run again.

★ **A runtime error happens during execution even though the code looks correct.**

4.4 Logical Errors

DEFINITION 4.4 — LOGICAL ERROR

A **logical error** occurs when the program runs to the end but gives the **wrong result**. There is no syntax problem and usually no error message: the computer does exactly what it was told, but it was told the wrong thing.

1 Common causes

- Wrong formula or calculation; wrong operator (`+ - * / %`).
- Wrong condition in `if` or a loop.
- Wrong algorithm; incorrect use of variables.

! Why they are difficult

- Python shows no message.
- The program looks correct and runs successfully.
- The bug is found only by checking the output carefully.

logic.py

```
1 total = 45                  # marks 10, 15, 20
2 avg = total / 2             # wrong: should divide by 3
3 print("Average:", avg)
4
5 length, breadth = 5, 3
6 area = length + breadth     # wrong: should multiply
7 print("Area:", area)
```

Output

```
Average: 22.5
Area: 8
```

Screen 4.3 Both lines run without any error, yet both answers are wrong (correct: 15.0 and 15).

EXAM POINT — FINDING LOGICAL ERRORS

Compare expected output with actual output · test with small inputs whose answer you know · check formulas, conditions, operators and variable use.

★ **A logical error lets the program run, but the result is wrong because the logic is wrong.**

4.5 Error Messages and Tracebacks

An **error message** tells *what* went wrong; the **traceback** shows *where* it happened and the path the program took to get there.

example.py

```
1 def average(marks):
2     return sum(marks) / len(marks)
3
4 print(average([80, 90]))
5 print(average([]))
```

Output

```
85.0
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    print(average([]))
          ~~~~~~~^^^^
  File "example.py", line 2, in average
    return sum(marks) / len(marks)
           ~~~~~~~~~~~^~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Screen 4.4 A traceback with a function call path: line 5 called `average`, and line 2 inside it raised the error.

1 Parts of a traceback

- **File** — the Python file where the error occurred.
- **Line** — the line number in that file, with the code line.
- **Call path** — how the code reached that line (most recent call last).
- **Error name and message** — e.g. `ZeroDivisionError: division by zero`.

2 How to read it

- Read the **last line first**: it gives the actual error type.
- Then look up to find the line number where it happened.
- That line (or the data reaching it) is where the problem is.

A traceback is like a map: it shows the path the program took and marks the spot of the mistake.

Table 4.3 Key error names

Term	Meaning
`SyntaxError`	Error in the structure (grammar) of the code
`NameError`	A name (variable or function) is used that is not defined
`TypeError`	An operation is applied to a value of the wrong type
`ZeroDivisionError`	Division (or `%`, `//`) by zero
Warning	A message about possible trouble; the program continues (not an error)

★ **Error messages and tracebacks tell the programmer the type and the location of a problem.**

4.6 Simple Debugging Techniques

DEFINITION 4.5 — DEBUGGING

Debugging means finding and correcting errors in a program. Good debugging is step by step, not random guessing.



1 Basic techniques

1. Read the error message carefully.
2. Check the line number shown.
3. Use `print()` to check values.
4. Test with small inputs.
5. Check variables and their values.

2 Other helpful methods

- Comment out parts of the code (`#`) to isolate the problem.
- Run one section at a time.
- Compare expected output with actual output.

wrong.py

```
1  a = 10
2  b = 5
3  sum = a + c      # c is not defined
4  print(sum)
```

Output

```
Traceback (most recent call last):
  File "wrong.py", line 3, in <module>
    sum = a + c      # c is not defined
              ^
NameError: name 'c' is not defined
```

fixed.py

```
1  a = 10
2  b = 5
3  print("a =", a, "b =", b)   # inspect values
4  total = a + b
5  print(total)
```

Output

```
a = 10 b = 5
15
```

Screen 4.5 Left: the `NameError` names the undefined variable. Right: the corrected program, with a `print()` used to inspect the values (and the name `total` instead of `sum`, so the built-in function `sum()` is not hidden).

GOOD HABITS

One change at a time, then test · save versions with different names · use meaningful variable names · run the program again after every change.

★ **Debugging is locating errors, correcting them and testing the program again.**

4.7 Exam Theory Notes

TRB EXAM THEORY — KEY FACTS

1. Three kinds of error: **syntax errors** (grammar; detected before running), **runtime errors / exceptions** (detected while running), **logical errors** (program runs but gives wrong output; no message).
2. `IndentationError` is a subclass of `SyntaxError`; `TabError` is a subclass of `IndentationError` (mixing tabs and spaces).
3. Common exceptions: `NameError` (undefined name), `TypeError` (wrong type), `ValueError` (right type, wrong value, e.g. `int("abc")`), `ZeroDivisionError`, `IndexError`, `KeyError`, `AttributeError`, `ImportError` / `ModuleNotFoundError`, `FileNotFoundError`, `RecursionError`, `OverflowError`.
4. All built-in exceptions derive from `BaseException`; normal ones derive from `Exception`. `KeyboardInterrupt` (Ctrl+C) and `SystemExit` derive directly from BaseException.
5. `ZeroDivisionError`, `OverflowError` and `FloatingPointError` are subclasses of `ArithmeticError`; `IndexError` and `KeyError` are subclasses of `LookupError`.
6. A **traceback** is read from the bottom: the last line names the exception and message; the lines above show the call chain with file names and line numbers.
7. Exceptions are handled with `try … except … else … finally`: `else` runs if no exception occurred; `finally` always runs.
8. `raise ValueError("msg")` raises an exception deliberately; `assert cond, "msg"` raises `AssertionError` when cond is False.
9. **Debugging** means finding and removing bugs. Techniques: reading tracebacks, printing intermediate values, tracing by hand (dry run), using a debugger.
10. The built-in debugger is `pdb`; `breakpoint()` (Python 3.7+) starts it at that line.
11. The term “bug” is popularly linked to a moth found in the Harvard Mark II computer in 1947 (logged by Grace Hopper's team).
12. Python 3.13 tracebacks underline the failing part with `^^^^` and suggest fixes such as “Did you mean: 'upper'?”.

PREDICT THE OUTPUT — EVERY RESULT VERIFIED BY RUNNING IT

`int("abc")`	`ValueError`	`10 / 0`	`ZeroDivisionError`
`[1, 2][5]`	`IndexError`	`{"a": 1}["b"]`	`KeyError`
`issubclass(IndentationError, SyntaxError)`	`True`	`issubclass(KeyError, LookupError)`	`True`
`issubclass(ZeroDivisionError, ArithmeticError)`	`True`	`"abc".push()`	`AttributeError`

Chapter 4 Summary

- Three kinds of error: syntax, runtime (exception) and logical.
- Syntax errors are found before anything runs.
- Runtime errors stop the program at the failing line.
- Logical errors give wrong output with no message.
- Read a traceback from the last line up: error type, then line and call path.
- Debug step by step: detect, understand, fix, test again.

Next — Chapter 5: values and data types.

CHAPTER · UNIT 1 BASICS

5 Values and Data Types

OPENING SITUATION

A student record holds a name, an age, an average mark, a pass/fail status and a list of subject marks. Each of these is a different kind of data, and Python treats each kind differently: you can add two ages, but adding a name to an age is an error. This chapter introduces values and the five basic data types of the syllabus — `int`, `float`, `bool`, `str` and `list` — with conversion and `type()`.

IN THIS CHAPTER

- 5.1 Meaning of Value
- 5.2 Integer Type
- 5.3 Float Type
- 5.4 Boolean Type
- 5.5 String Type
- 5.6 List Type
- 5.7 Type Conversion Basics
- 5.8 Checking Type using type()
- 5.9 Exam Theory Notes

YOU NEED

- Chapters 3–4: running programs and reading errors.

5.1 Meaning of Value

DEFINITION 5.1 — VALUE AND DATA TYPE

A **value** is the actual data or information used in a program, such as `10`, `3.14` or `"Python"`. Every value belongs to a **data type**, which decides what the value means and which operations are allowed on it.

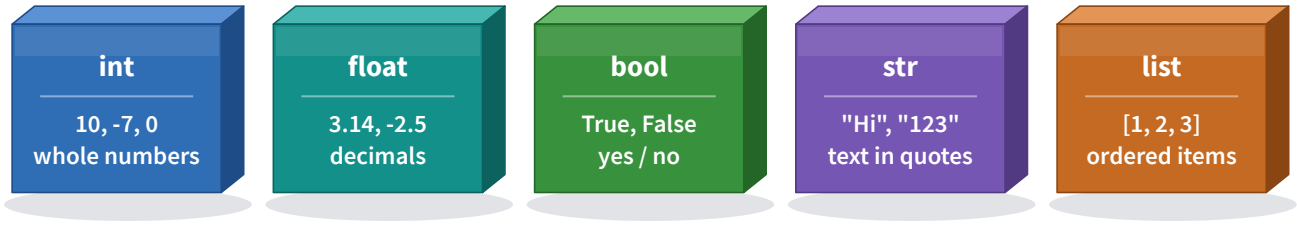


Figure 5.1 The five basic data types of the syllabus, with example values and the type name that `type()` reports.

1 Main idea

- Every piece of data in Python is a value, and every value has a type.
- Values can be stored in variables and used in expressions.
- Programs take values as input, process them and produce values as output.

2 In simple words

A value is like the content kept inside a box. In a program, the box may be a variable, and the content inside it is the value. Real-life values: marks `95`, name `"Ravi"`, status `True`.

values.py

```
1  x = 25
2  name = "Anu"
3  print(x)
4  print(name)
```

Output

```
25
Anu
```

★ **A value is the actual data that a program stores, uses and processes.**

5.2 Integer Type

DEFINITION 5.2 — INT

The **integer type** (`int`) represents whole numbers without a decimal point: positive, negative or zero, such as `10`, `-7`, `0`, `2024`. Integers are used for counting and exact quantities — students in a class, marks, age, number of items.

integers.py

```
1  a = 25
2  b = -10
3  c = 0
4  print(type(a))
5  print(a + b, a * b, c)
6  print(2 ** 100)          # Python integers have no fixed size limit
```

Output

```
<class 'int'>
15 -250 0
1267650600228229401496703205376
```

Screen 5.1 Integers can be positive, negative or zero. Python integers grow as large as memory allows — `2 ** 100` is exact.

★ An integer stores whole numbers such as −10, 0, 25 and 125.

5.3 Float Type

DEFINITION 5.3 — FLOAT

The **float type** represents numbers with a decimal point, such as `3.14`, `0.75`, `-2.5` and `12.0`. Floats are used for measured values and fractions: average marks, height, weight, price, temperature, scientific values.

floats.py

```
price = 99.50
avg = 87.6
x = 10 / 4              # / always gives a float
print(type(price), x)
print(12.0 == 12, 1.5e3)
print(0.1 + 0.2)
```

▶ Output

```
<class 'float'> 2.5
True 1500.0
0.30000000000000004
```

Screen 5.2 `/` always returns a float. `1.5e3` is scientific notation for 1500.0. Floats are stored in binary, so `0.1 + 0.2` is very slightly off — use `round()` when displaying.

★ A float stores decimal numbers such as 3.14, 0.75 and 99.50.

5.4 Boolean Type

DEFINITION 5.4 — BOOL

The **Boolean type** (`bool`) has only two values, `True` and `False` (capital first letter). Booleans are the results of comparisons and are used in conditions and decision making — like a switch that is either ON (`True`) or OFF (`False`).

booleans.py

```
is_pass = True
is_even = False
print(10 > 5, 3 == 7)
print(type(is_pass))
```

▶ Output

```
True False
<class 'bool'>
```

Uses

`if` statements, checking conditions, yes/no decisions, controlling program flow.

Real life

Login valid or invalid · pass or fail · light on or off · true/false answers.

Note

`true` (small t) is not a Boolean — it is an undefined name.

★ A Boolean stores True or False and helps Python make decisions.

5.5 String Type

DEFINITION 5.5 — STR

The **string type** (`str`) represents text: a sequence of characters enclosed in single or double quotes, such as `'Python'` , `"Hello"` , `'Ravi'` . A string may be one letter, a word, a sentence, or digits treated as text.

strings.py

```
1 name = 'Anu'
2 msg = "Welcome"
3 print(name)
4 print(type(msg))
5 print("123" + "4", 123 + 4)
```

Output

```
Anu
<class 'str'>
1234 127
```

Screen 5.3 `"123"` is text, not a number: `+` joins strings but adds integers. Names, city names, greetings and even mobile numbers are stored as strings.

★ A string stores text data written inside quotes.

5.6 List Type

DEFINITION 5.6 — LIST

A **list** is an ordered collection of values stored in one variable, written inside square brackets with items separated by commas. Items may be of the same type or of mixed types. Positions (indexes) start at **0**.

lists.py

```
1 marks = [90, 85, 78]
2 mixed = [10, 'Hi', True]
3 print(marks[0])
4 print(type(marks), len(mixed))
```

Output

```
90
<class 'list'> 3
```

Screen 5.4 `marks[0]` is the first item. A list is like a box that holds many values in order; you can look at any value by its position. Examples: marks `[95, 88, 76, 91]` , names `['Ravi', 'Anu', 'Priya']` , shopping items.

★ A list stores multiple ordered values in one variable using square brackets.

5.7 Type Conversion Basics

DEFINITION 5.7 — TYPE CONVERSION

Type conversion (type casting) means changing a value from one data type to another with the built-in functions `int()` , `float()` , `str()` and `bool()` . It is needed mainly because `input()` always returns a string.

Table 5.1 Conversion functions

Function	Example	Result	Note
`int()`	`int('10')`, `int(3.9)`	`10`, `3`	Float → int drops the fraction (no rounding)
`float()`	`float('3.14')`	`3.14`	
`str()`	`str(25)`	`'25'`	Number becomes text
`bool()`	`bool(0)`, `bool(5)`	`False`, `True`	Zero and empty values are False

convert.py

```
1 age = input("Enter age: ")      # input is always text
2 print(type(age))
3 age = int(age)
4 print("Next year:", age + 1)
5 print(float('99.5'), str(100), int(-3.9))
6 print(bool(0), bool(""), bool("False"))
```

Output

```
Enter age: 20
<class 'str'>
Next year: 21
99.5 100 -3
False False True
```

Screen 5.5 Conversion makes input usable in calculations. `int(-3.9)` gives `-3` (the fraction is cut off toward zero).

COMMON MISTAKE 5.1 — BOOL("FALSE") IS FALSE

`bool()` of any **non-empty** string is `True` — even the string `"False"`. Only the empty string `""` converts to `False`. Invalid conversions such as `int('abc')` raise `ValueError`.

★ Type conversion changes a value from one type to another when needed.

5.8 Checking Type using type()

The built-in `type()` function tells you the data type of a value or variable. It is useful in learning and debugging — to confirm a type before a calculation, to check that input was converted, and to understand how Python handles different values.

>>> Python Shell

```
>>> type(10)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type('Hi')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type([1, 2])
<class 'list'>
>>> isinstance(10, int)
True
```

Screen 5.6 `type()` reports the class of each value; `isinstance(value, type)` answers yes/no.

★ type() displays the data type of a value or variable.

5.9 Exam Theory Notes

TRB EXAM THEORY — KEY FACTS

1. A **value** is a basic piece of data (5, 3.2, "hi"); every value is an **object** with an identity (`id`), a type (`type`) and a value.
2. Built-in types: numeric — `int` , `float` , `complex` ; Boolean — `bool` ; sequence — `str` , `list` , `tuple` , `range` ; mapping — `dict` ; set — `set` , `frozenset` ; binary — `bytes` , `bytearray` ; `NoneType` .
3. **int** has unlimited precision (arbitrary size); there is no separate `long` type in Python 3.
4. Integer literals: decimal `25` , binary `0b11001` , octal `0o31` , hexadecimal `0x19` ; underscores group digits: `1_000_000` .
5. **float** is a 64-bit IEEE 754 double (about 15–17 significant digits); scientific notation `1.5e3` = 1500.0. Floats are approximate: `0.1 + 0.2 != 0.3` .
6. **complex** numbers are written with `j` : `3 + 4j` ; attributes `.real` , `.imag` ; method `.conjugate()` ; `abs(3+4j)` is 5.0.
7. **bool** is a subclass of `int` : `True == 1` and `False == 0` ; `True + True` is 2.
8. `None` is the only value of `NoneType` ; it means “no value” and is returned by functions without a return statement.
9. **Mutable** types: list, dict, set, bytearray. **Immutable** types: int, float, complex, bool, str, tuple, frozenset, bytes.
10. Conversion functions: `int()` , `float()` , `complex()` , `str()` , `bool()` , `list()` , `tuple()` , `set()` , `dict()` ; base conversion `bin()` , `oct()` , `hex()` , `int("ff", 16)` .
11. `int()` truncates towards zero: `int(-3.9)` is −3; `round()` uses **banker's rounding** (round half to even): `round(2.5)` is 2.
12. **Implicit** conversion (coercion): int + float gives float. **Explicit** conversion (type casting) uses the functions above.
13. `type(x)` returns the type; `isinstance(x, int)` tests the type and also accepts subclasses, so `isinstance(True, int)` is True.
14. Python has no separate character type: a character is a string of length 1.

PREDICT THE OUTPUT — EVERY RESULT VERIFIED BY RUNNING IT

`0b1010 + 0o12 + 0xA`	`30`	`int("ff", 16)`	`255`
`bin(10), oct(10), hex(255)`	`('0b1010', '0o12', '0xff')`	`0.1 + 0.2 == 0.3`	`False`
`True + True`	`2`	`int(-3.9)`	`-3`
`round(2.5), round(3.5)`	`(2, 4)`	`abs(3 + 4j)`	`5.0`
`type(None)`	`<class 'NoneType'>`	`isinstance(True, int)`	`True`
`1_000_000 // 1000`	`1000`	`2 ** 100 > 10 ** 30`	`True`
`float("inf") > 10 ** 308`	`True`	`complex(2)`	`(2+0j)`

Chapter 5 Summary

- Every value has a data type that decides its allowed operations.
- `int` whole numbers (unlimited size); `float` decimals.
- `bool` is `True` / `False` ; results of comparisons.
- `str` is text in quotes; `"123"` is not a number.
- `list` is an ordered collection in `[ ]` , index from 0.
- `int()` , `float()` , `str()` , `bool()` convert; `type()` checks.

Next — Chapter 6: variables.

CHAPTER · UNIT 1 BASICS

6 Variables in Python

OPENING SITUATION

The shop-billing program in the lab needs the price, the quantity and the total, and it uses each of them several times. Instead of typing the same numbers again and again, the program gives each value a name. These names are variables. This chapter covers what a variable really is in Python, the naming rules, assignment, multiple assignment and dynamic typing.

IN THIS CHAPTER

- 6.1 Meaning of Variable
- 6.2 Need for Variables
- 6.3 Variable Naming Rules
- 6.4 Assigning Values to Variables
- 6.5 Multiple Assignment
- 6.6 Dynamic Typing in Python
- 6.7 Common Variable Naming Mistakes
- 6.8 Exam Theory Notes

YOU NEED

- Chapter 5: values and data types.

6.1 Meaning of Variable

DEFINITION 6.1 — VARIABLE

A **variable** is a **name that refers to a value** (an object) stored in memory. The name is used to store, reuse and change data in a program; the value it refers to can change during execution.

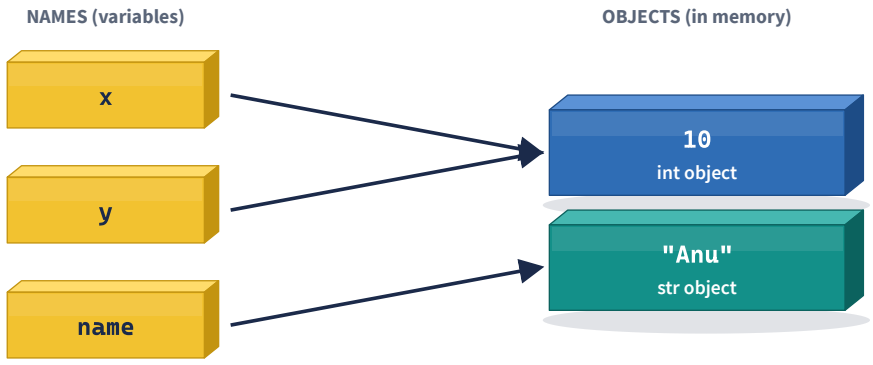


Figure 6.1 Python variables are name tags attached to objects. `x = 10` creates the object `10` and binds the name `x` to it; after `y = x` both names refer to the same object.

1 Main idea

- Variables hold data that can be used and reused.
- They make programs readable and flexible.
- A variable can refer to values of any type.
- Names are **case-sensitive**: `name`, `Name` and `NAME` are three different variables.

2 In simple words

A variable is like a labelled box (or a name tag): keep something inside, and whenever you need that value, just use the label.

variables.py

```
1 x = 10
2 name = "Anu"
3 is_pass = True
4 print(x, name, is_pass)
```

Output

```
10 Anu True
```

COMMON MISTAKE 6.1 — "A VARIABLE IS A FIXED MEMORY BOX OF ONE TYPE"

In C a variable is a typed memory location. In Python the *object* has the type and lives in memory; the variable is only a reference to it. Assigning a new value makes the name refer to a new object — it does not overwrite the old box.

★ **A variable is a name used to store and refer to values, making code readable, reusable and flexible.**

6.2 Need for Variables

Variables are needed to store data so that a program can use it again later instead of writing the same value repeatedly. They store information, make calculations easier, let values change during execution and improve readability.

1 Store

`marks = 95`, `name = "Anu"` — keep marks and text.

2 Calculate

`total = 250` — results of calculations.

3 Change

`count = 1` then `count = count + 1` — counters and totals.

bill.py

```
1 price = 50
2 quantity = 3
3 total = price * quantity
4 print("Total:", total)
5 quantity = 4                  # update once, reuse everywhere
6 print("New total:", price * quantity)
```

Output

```
Total: 150
New total: 200
```

Screen 6.1 Variables reduce repetition, support calculations, store user input and make updates easy — student marks, salaries, prices and counters all use them.

★ **Variables are needed to store, reuse and update data easily.**

6.3 Variable Naming Rules

Rules (must)

- May contain letters, digits and underscore `_`.
- Must begin with a letter or underscore — never a digit.
- No spaces or special symbols (`- @ $ %` ...).
- Cannot be a keyword (`class`, `if`, `for` ...).
- Case-sensitive: `total` ≠ `Total`.

Good style (should)

- Use meaningful names: `total_marks`, not `t`.
- Use **snake_case** (lower case with underscores).
- Avoid built-in names such as `sum`, `list`, `print`.

Table 6.1 Valid and invalid names

Valid	Invalid	Why invalid	Correct
`age`	`1name`	Starts with a digit	`name1`
`total_marks`	`total marks`	Contains a space	`total_marks`
`student1`	`my-name`	`-` means minus	`my_name`
`_count`	`class`	Keyword	`student_class`

>>> Python Shell

```
>>> 1name = "Anu"
  File "<stdin>", line 1
    1name = "Anu"
    ^
SyntaxError: invalid decimal literal
>>> class = 5
  File "<stdin>", line 1
    class = 5
          ^
SyntaxError: invalid syntax
>>> import keyword
>>> print(len(keyword.kwlist), keyword.kwlist[:12])
35 ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def']
```

Screen 6.2 Invalid names are syntax errors. `keyword.kwlist` lists the reserved keywords that cannot be used as names.

★ **Variable names must follow Python's rules and should be clear, meaningful and readable.**

6.4 Assigning Values to Variables

DEFINITION 6.2 — ASSIGNMENT

Assignment stores a value in a variable using the `=` operator. The left side is the variable name; the right side is a value or expression. Python **evaluates the right side first**, then binds the name to the result. A variable is created the first time a value is assigned to it.

SYNTAX

```
variable_name = value_or_expression
```

assign.py

```
x = 5
y = 10
total = x + y        # right side evaluated first
print(total)
x = x + 1            # old value replaced by a new value
print(x)
```

Output

```
15
6
```

COMMON MISTAKE 6.2 — = AND ==

`=` is **assignment** (store a value); `==` is **comparison** (are two values equal?). `x = x + 1` is not a false equation; it means "compute `x + 1`, then make `x` refer to the result".

★ Assignment stores a value or result inside a variable using the = operator.

6.5 Multiple Assignment

Multiple assignment assigns values to several variables in one statement. It saves time, reduces repeated code and makes initialisation easy. The number of values must match the number of names, and order matters.

Table 6.2 Forms of multiple assignment

Statement	Meaning
`a, b, c = 1, 2, 3`	1 to `a`, 2 to `b`, 3 to `c` (tuple assignment)
`x = y = z = 0`	The same value to all three names
`name, age = "Anu", 18`	Values of different types together
`a, b = b, a`	Swap two values without a temporary variable

multi.py

```
x, y, z = 5, 10, 15
print("Total:", x + y + z)
a = b = c = 0
print(a, b, c)
p, q = 1, 2
p, q = q, p          # swap
print(p, q)
m, n = 1, 2, 3       # count mismatch
```

Output

```
Total: 30
0 0 0
2 1
Traceback (most recent call last):
  File "multi.py", line 8, in <module>
    m, n = 1, 2, 3       # count mismatch
    ^^^^
ValueError: too many values to unpack (expected 2)
```

Screen 6.3 Values must match the variables exactly; three values for two names raises `ValueError`. Tuple assignment is studied fully in Chapter 7 and Part VII.

★ Multiple assignment gives values to several variables in a single line.