MITE

Invent Solutions

Mangalore Institute of Technology & Engineering

Moodabidri, Karnataka

Department of Artificial Intelligence and Machine Learning

Modern Computing and Python Programming

Course Code: 26ESCC106

First Year B.E. (Physics Cycle) • Common to all branches

Module 2

Python Fundamentals

Study Notes

Topics Covered
1. About Python and Program Structure
2. Variables, Keywords, Identifiers and Literals
3. Input and Output Statements
4. Data Types, Operators and Expressions
5. Operator Precedence and Type Conversion
6. Basic Debugging Techniques
7. Laboratory Programs

Text Book: Reema Thareja, *Python Programming using Problem Solving Approach*, Oxford University Press, 3rd Edition, 2024 (Sections 3.1, 3.2, 3.3, 3.5, 3.6, 3.7, 3.12, 3.16; supporting: 3.4, 3.8–3.11, Annexure 4).

Contents

Topic 1: About Python and Program Structure **4**

1 Introduction to Python and Its Features 4

2 History, Applications, Future, and Limitations of Python 5

3 Writing and Executing a Python Program 6

4 Structure of a Python Program: Comments, Indentation, and Statements 6

5 Question Bank 8

Topic 2: Variables, Keywords, Identifiers and Literals **9**

1 Literal Constants 9

2 Variables, Identifiers, and Keywords 10

3 Question Bank 11

Topic 3: Input and Output Statements **12**

1 Input and Output Statements 12

2 Question Bank 12

Topic 4: Data Types, Operators and Expressions **13**

1 Data Types and Assignment 13

2 Operators 14

3 Expressions 15

4 Question Bank 16

Topic 5: Operator Precedence and Type Conversion **17**

1 Operator Precedence and Associativity 17

2 Type Conversion 18

3 Question Bank 19

Topic 6: Basic Debugging Techniques **20**

1 Basic Debugging Techniques 20

2 Question Bank 21

Topic 7: Laboratory Programs **22**

1 Program 1: Student Details, Total, and Average . 22

2 Program 2: Area and Circumference of a Circle . 22

3 Program 3: Values and Data Types 23

4 Program 4: Arithmetic Operations on Two Numbers 24

How to Use These Notes

- The notes follow the order of the Module 2 syllabus. Each topic starts on a new page and its sections are numbered from 1.
- Each section ends with a typical university-style question and its marks (6 or 8); the content above it is written so that it can be reproduced as the answer.
- **6-mark answer:** definition + 5–6 explained points + one short example program or diagram.
- **8-mark answer:** introduction + 6–8 explained points + example programs with output (or a neat diagram) + a concluding sentence.
- All programs are written for **Python 3** and their outputs have been verified by running them. Where the textbook shows Python 2 syntax, the Python 3 form is used and the difference is noted.
- Topics 1–6 end with a question bank for practice; Topic 7 contains the four laboratory programs of this module.

Note on Preparation

AI assistance was used to draft and to typeset these notes in LaTeX. The content follows the prescribed textbook and the syllabus scope of the course, and has been reviewed by the course coordinator.

Module 2: Python Fundamentals

Topic 1: About Python and Program Structure

Features, history, applications, future and limitations of Python; writing and executing a Python program; structure of a program — comments, indentation and statements

1. Introduction to Python and Its Features

Python is a **high-level, interpreted, interactive, object-oriented**, and reliable programming language that is very simple and uses English-like words. It has a vast library of modules to support the building of complex solutions from pre-built components.

Python is an open-source project supported by many individuals. It is a platform-independent, scripted language with complete access to operating system APIs.

Features of Python:

1. **Simple:** Reading a Python program feels almost like reading English. This lets programmers concentrate on the solution to the problem rather than on the language.
2. **Easy to learn:** A Python program is clearly defined and easily readable. It uses few keywords and a clearly defined syntax.
3. **Versatile:** Python supports the development of a wide range of applications, from simple text processing to web browsers and games.
4. **Free and open source:** Anyone can freely distribute Python, read its source code, edit it, and use it to write new programs.
5. **High-level language:** Programmers need not worry about low-level details such as managing the memory used by the program.
6. **Interactive:** Programs can be written and tested interactively at the Python prompt, which allows easy testing and debugging of pieces of code.
7. **Portable:** Programs behave the same on a wide variety of hardware platforms and operating systems (Linux, Windows, macOS, …) without changes.
8. **Object oriented:** Python supports both object-oriented and procedure-oriented styles of programming.
9. **Interpreted:** Python is processed at run time by the interpreter; there is no need to compile a program before executing it. Python converts the source code into an intermediate form called **bytecode**, which is then translated into the native language of the computer.
10. **Dynamic:** Programs are executed dynamically; errors are reported at run time, which allows interactive program development.
11. **Extensible and embeddable:** Low-level modules written in C or C++ can be added to the interpreter, and Python can be embedded in C, C++, Java, and other programs to give scripting capabilities.
12. **Extensive libraries:** A huge library that is portable across platforms, for tasks such as text processing, databases, and GUI programming.
13. **Robust and secure:** Errors are raised as exceptions that can be caught and handled; modules can be distributed to prevent tampering.
14. **Multi-threaded and garbage collected:** Python can execute several parts of a program at once, and it frees the memory of unused objects automatically.

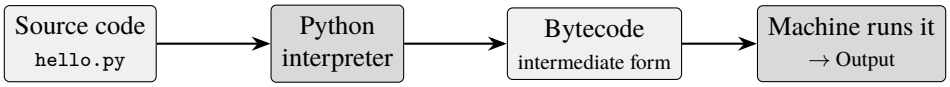


Figure 1.1: How Python runs a program — the interpreter converts source code into bytecode

Conclusion: Because of these features, Python is used by companies such as Google, YouTube, and NASA, and is the leading language for AI and machine learning.

Exam question on this section:

Q. What is Python? Explain any eight features of Python. **[8 Marks]**

2. History, Applications, Future, and Limitations of Python

History of Python:

1. Python was developed by **Guido van Rossum** in the late 1980s and early 1990s at the **National Research Institute for Mathematics and Computer Science (CWI)** in the Netherlands.
2. It was derived from many languages, such as ABC, Modula-3, C, C++, Algol-68, SmallTalk, and the Unix shell.
3. It was released in **February 1991**. Python 2.0 was released in 2000, and **Python 3** was released in 2008. Python 3 is the version in use today.
4. Its source code is freely available, and it is maintained by a core development team.
5. **Name:** Guido was reading the scripts of **Monty Python's Flying Circus**, a BBC comedy series, and wanted a short, unique, and slightly mysterious name.

Applications of Python:

Area	Examples
Web development	Django, Pyramid, Flask, Bottle; Quora, Reddit, Google App Engine
Scientific and computational	Scientific Python, NumPy, SciPy, Pandas; FreeCAD; Abaqus
GUI desktop applications	wxPython, PyQt, PyGtk
Image processing and graphic design	GIMP, Inkscape, Paint Shop Pro, Blender, Maya
Games	PyGame, PySoy; Civilization-IV, Toontown Online
Operating systems	Ubuntu's Ubiquity installer; Red Hat's Anaconda installer
Others	Embedded scripting, 3D software, prototyping, network programming, teaching, language development

Exam question on this section:

Q. Write a short note on the history and applications of Python. **[6 Marks]**

Future of Python:

1. Python has a huge user base that is constantly growing; it is a stable language that is here to stay.
2. It is a preferred language of companies such as Nokia, Google, and YouTube, and of NASA, for its easy syntax.
3. It supports multiple programming paradigms — object-oriented, functional, and parallel programming.
4. It is among the top five most preferred languages in academia and industry.

Limitations of Python:

1. Being interpreted, Python is **slower than C/C++**; it is not a good choice for high-graphic 3D games that use a lot of CPU.

2. Parallel processing is not as elegant as in some other languages, and it lacks true multiprocessor support.
3. Compared with C, C++, or Java there were fewer users, and commercial support is limited.
4. It is evolving continuously, so substantial documentation may be lacking.
5. It is difficult to package a big Python application into a single executable file for non-technical users.

Exam question on this section:

> **Q.** Explain the future and the limitations of Python. **[6 Marks]**

3. Writing and Executing a Python Program

Python is downloaded from `www.python.org`. It comes with **IDLE** (Integrated Development and Learning Environment), which provides a shell window, an interactive interpreter, a debugger, and a multi-window text editor with features such as colouring, smart indentation, and auto-completion.

Steps to write and run a Python program:

1. **Step 1:** Open an editor (in IDLE: **File** → **New File**).
2. **Step 2:** Write the instructions.
3. **Step 3:** Save the file with the extension `.py`.
4. **Step 4:** Run the program — press **F5** (or **Run** → **Run Module**) in IDLE, or type `python program_name.py` at the command prompt.

To exit IDLE: **File** → **Exit**, **Ctrl + Q**, or type `quit()`.

```
>>> print("Hello World!!!")
Hello World!!!
```

Sl.	Aspect	Interactive Mode	Script Mode
1	Where	Python shell, at the >>> prompt	A `.py` file in an editor
2	Execution	One statement at a time	The whole program at once
3	Result	Shown immediately	Shown after running (F5)
4	Saving	Not saved	Saved and can be run again
5	Use	Testing small pieces of code	Writing complete programs

Exam question on this section:

> **Q.** Explain the steps to write and execute a Python program. Differentiate interactive mode and script mode. **[6 Marks]**

4. Structure of a Python Program: Comments, Indentation, and Statements

Structure of a Python program: Most programs follow **Input** → **Processing** → **Output**. Statements are executed one after another from top to bottom.

```
# Program to find the area of a rectangle      <- comments
# Author: Student of MITE

length = float(input("Enter length: "))      # input
width  = float(input("Enter width: "))

area = length * width                         # processing
```

```
print("Area =", area)                          # output
```

Output

```
Enter length: 5
Enter width: 4
Area = 20.0
```

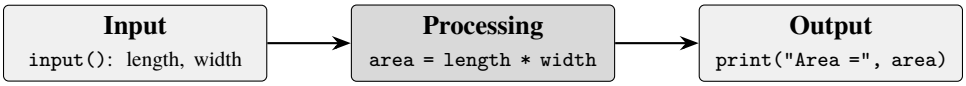


Figure 1.2: Input–Processing–Output structure of a program

1. Comments

- Comments are **non-executable** statements added to describe the program. They make the program easily readable for the programmer and for others.
- In Python, a hash sign (#) that is not inside a string begins a comment; everything after it up to the end of the line is ignored by the interpreter.
- A comment can be on its own line or after a statement. A program can have any number of comments.

2. Indentation

- Whitespace (spaces and tabs) at the beginning of a line is called **indentation**.
- In most languages indentation only improves readability; in Python it is used to **associate and group statements into blocks**. Python does not use curly braces for blocks.
- All statements inside a block must have the **same indentation level**; otherwise Python reports an error.

```
age = 21
    print("You can vote")
```

Output

```
    print("You can vote")
IndentationError: unexpected indent
```

3. Statements: physical and logical lines

- A **physical line** is what we see in the editor; a **logical line** is what Python sees as a single statement.
- It is good practice to write **one statement per line**.
- More than one statement on a line must be separated by a semicolon: `msg = "Hello"; print(msg)`. This should be avoided.

Conclusion: Comments make a program understandable, indentation gives it structure, and one-statement-per-line keeps it readable.

Exam question on this section:

Q. Explain the structure of a Python program. Explain comments, indentation, and statements with examples.

[8 Marks]

5. Question Bank

Sl.	Question	Marks	Section
1	What is Python? Explain any eight features of Python.	8	1
2	Write a short note on the history and applications of Python.	6	2
3	Explain the future and the limitations of Python.	6	2
4	Explain the steps to write and execute a Python program. Differentiate interactive and script mode.	6	3
5	Explain the structure of a Python program. Explain comments, indentation, and statements.	8	4

Module 2: Python Fundamentals

Topic 2: Variables, Keywords, Identifiers and Literals

Literal constants — numbers, strings and escape sequences; variables and identifiers; rules for naming identifiers; keywords of Python 3

1. Literal Constants

A **literal constant** is a value used directly in a program, such as `7`, `3.9`, `'A'`, and `"Hello"`. The number `7` always represents itself (*literal*), and its value cannot be changed (*constant*).

1. Numbers

- **Integers:** whole numbers such as `5`. Python integers can be of any size. (Python 2 had a separate *long integer* type with suffix `L`; in Python 3 all are `int`.)
- **Floating point numbers:** numbers with a decimal point such as `3.23` and `91.5E-2` (`E` means a power of 10: 91.5×10^{-2}).
- **Complex numbers:** numbers of the form $a + bj$, such as `-3 + 7j`.
- Commas are never used in numeric literals: `3,567` is not allowed.

Issues with floating point numbers:

- **Arithmetic overflow:** the result is too large; e.g., `2.7e200 * 4.3e200` gives `inf`.
- **Arithmetic underflow:** the result is too small; e.g., `3.0e-400 / 5.0e200` gives `0.0`.
- **Loss of precision:** e.g., `1/3` gives `0.3333333333333333`, only an approximation.

```
>>> 96 / 12              # / always gives a float
8.0
>>> 78 // 5, 78 % 5      # quotient and remainder
(15, 3)
>>> 5 ** 3               # exponentiation
125
>>> format(16/3, '.2f')  # 2 decimal places
'5.33'
```

2. Strings

A **string** is a group of characters. Strings can be written using:

- **Single quotes:** `'HELLO'`
- **Double quotes:** `"HELLO"` — exactly the same as single quotes.
- **Triple quotes:** `'''…'''` — for multi-line strings.

Strings are **immutable** (cannot be changed once created). String literals placed side by side are concatenated automatically. Python has no separate character data type.

3. Escape sequences

An **escape sequence** is a combination of a backslash and a character that represents a character which is difficult or impossible to type directly.

Escape	Purpose	Example	Output
\\	Prints backslash	`print("\\")`	\
\'	Prints single quote	`print("\'")`	'
\"	Prints double quote	`print("\"")`	"
\n	New line	`print("Hello\nWorld")`	Hello / World (2 lines)
\t	Tab	`print("Hello\tWorld")`	Hello World
\a	Bell	`print("\a")`	(beep)

```
>>> print('What\'s your name?')
What's your name?
>>> print(r"What\'s your name?")     # raw string
What\'s your name?
```

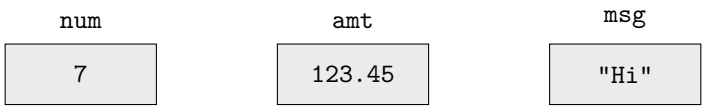
A **raw string**, prefixed with `r` or `R`, does not process escape sequences.

Exam question on this section:

Q. What are literal constants? Explain numeric and string literals with examples. Explain escape sequences. **[8 Marks]**

2. Variables, Identifiers, and Keywords

A **variable** is a **reserved memory location** that stores a value. Its value can vary while the program runs. An **identifier** is a name given to identify a variable, function, class, module, or other object. Variables are examples of identifiers.



Each variable is a labelled box in memory: the name is the label, the value is inside.

Figure 2.1: Variables as named memory locations

Rules for naming identifiers:

1. The first character must be an underscore (_) or a letter (upper or lower case).
2. The rest of the name can contain underscores, letters, or digits (0–9).
3. Identifier names are **case-sensitive**: `myvar` and `myVar` are different.
4. Punctuation characters such as `@`, `$`, and `%` are not allowed.
5. A keyword cannot be used as an identifier.

Valid identifiers: `sum`, `__my_var`, `num1`, `r`, `var_20`, `First`
Invalid identifiers: `1num`, `my-var`, `%check`, `Basic Sal`, `H#R&A`

Exam question on this section:

Q. What is a variable? What is an identifier? State the rules for naming identifiers with valid and invalid examples. **[6 Marks]**

Keywords (reserved words) are words with a **pre-defined meaning** in the language. They cannot be used for naming identifiers. Python 3.11 has **35 keywords**:

`False`	`None`	`True`	`and`	`as`	`assert`	`async`
`await`	`break`	`class`	`continue`	`def`	`del`	`elif`
`else`	`except`	`finally`	`for`	`from`	`global`	`if`
`import`	`in`	`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`	`with`	`yield`

All keywords contain lowercase letters only, **except** `True`, `False`, and `None`. The list can be displayed with `import keyword; print(keyword.kwlist)`. (Table 3.2 of the textbook is from Python 2, where `print` and `exec` were also keywords.)

Exam question on this section:

Q. What are keywords? List the keywords of Python 3. **[6 Marks]**

3. Question Bank

Sl.	Question	Marks	Section
1	What are literal constants? Explain numeric and string literals and escape sequences.	8	1
2	What is a variable? What is an identifier? State the rules for naming identifiers.	6	2
3	What are keywords? List the keywords of Python 3.	6	2

Module 2: Python Fundamentals

Topic 3: Input and Output Statements

Displaying output with `print()` and `format()`; reading input with `input()`; converting input to numbers

1. Input and Output Statements

Output: `print()`

- `print()` displays values on the screen. Several values separated by commas are printed with a space between them.
- `sep` sets the separator between values; `end` sets what is printed at the end (default: new line).
- `format(value, specifier)` formats a value, e.g., `'.2f'` for 2 decimal places, `'<10'`, `'>10'`, `'^10'` for left, right, and centre alignment.

```
print("Total =", 250)
print(1, 2, 3, sep="-")
print("Average =", format(85.3333, ".2f"))
```

Output

```
Total = 250
1-2-3
Average = 85.33
```

Input: `input()`

- `input(prompt)` displays the prompt, waits for the user to type, and returns the typed text.
- `input()` **always returns a string**, even if a number is typed. (In Python 2 it was called `raw_input()`.)
- To read numbers, convert the input: `int(input(…))` or `float(input(…))`.

```
name = input("What's your name? ")
age = input("Enter your age : ")
print(name + ", you are " + age + " years old")
```

Output

```
What's your name? Goransh
Enter your age : 10
Goransh, you are 10 years old
```

Exam question on this section:

Q. Explain the input and output statements in Python with examples. **[6 Marks]**

2. Question Bank

Sl.	Question	Marks	Section
1	Explain the input and output statements in Python with examples.	6	1
2	Explain the `sep` and `end` parameters of `print()` and the `format()` function with examples.	6	1
3	Why does `input()` always return a string? How are numbers read from the keyboard? Explain with an example.	6	1

Module 2: Python Fundamentals

Topic 4: Data Types, Operators and Expressions

Data types and assignment; arithmetic, comparison, assignment, logical, unary, bitwise, shift, membership and identity operators; types of expressions

1. Data Types and Assignment

Variables can hold values of different types called **data types**. Based on the data type of a variable, the interpreter reserves memory for it and decides what data can be stored in it. The basic types are **numbers** and **strings**; Python's standard data types include numbers, string, list, tuple, and dictionary.

Data Type	Name	Example	`type()` output
Integer	`int`	`7`	`<class 'int'>`
Floating point	`float`	`3.9`	`<class 'float'>`
Complex	`complex`	`3+4j`	`<class 'complex'>`
String	`str`	`'A'`	`<class 'str'>`
Boolean	`bool`	`True`	`<class 'bool'>`

1. Assigning values: Variables need not be declared. A variable is created when a value is assigned using `=`; the name on the left gets the value on the right.

2. Reassigning values: A variable can be reassigned as many times as needed, even with a value of a different type, because Python variables do not have fixed types.

```
val = 'Hello'
print(val)
val = 100
print(val)
val = 12.34
print(val)
```

Output

```
Hello
100
12.34
```

3. Multiple assignment: One value can be assigned to several variables, or several values to several variables, in one statement.

```
sum = flag = a = b = 0
sum, a, b, mesg = 0, 3, 5, "RESULT"
```

4. Undefined variables: Using a variable that has not been assigned, or that was deleted with `del`, gives a `NameError`.

```
>>> print(age)
NameError: name 'age' is not defined
```

5. Boolean: A Boolean variable has one of two values, `True` or `False`. Boolean values are produced when values are compared.

```
>>> 20 == 30
False
>>> 87 == 87.0
True
>>> "Python" == "Python"
```

```
True
```

Conclusion: Python decides the data type automatically from the value assigned; `type()` shows it.

Exam question on this section:

> **Q.** Explain the data types in Python. Explain assignment, reassignment, multiple assignment, and the Boolean data type with examples. **[8 Marks]**

2. Operators

> **Operators** are constructs used to manipulate the values of **operands**. In the expression `sum = a + b`, `a` and `b` are operands and `+` is the operator.

Python supports arithmetic, comparison (relational), assignment, logical, unary, bitwise, membership, and identity operators.

1. Arithmetic operators (with `a = 100`, `b = 200`):

Op.	Description	Example	Output
`+`	Addition	`a + b`	`300`
`-`	Subtraction	`a - b`	`-100`
`*`	Multiplication	`a * b`	`20000`
`/`	Division; always returns a float	`b / a`	`2.0`
`%`	Modulus; returns the remainder	`b % a`	`0`
`//`	Floor division; quotient rounded towards negative infinity	`12 // 5`, `-19 // 5`	`2`, `-4`
`**`	Exponent; raises left operand to the power of the right	`a ** 2`	`10000`

2. Comparison (relational) operators compare values and return `True` or `False`: `==` (equal), `!=` (not equal), `>`, `<`, `>=`, `<=`. With `a = 100`, `b = 200`: `a == b` is `False`, `a != b` is `True`, `a < b` is `True`.

3. Assignment and in-place (shortcut) operators: `=` assigns a value. Shortcut operators combine an operation with assignment:

`a += b`	`a -= b`	`a *= b`	`a /= b`	`a %= b`	`a //= b`	`a **= b`
`a = a + b`	`a = a - b`	`a = a * b`	`a = a / b`	`a = a % b`	`a = a // b`	`a = a ** b`

In-place operators also work on strings: if `str1 = "Good "`, then `str1 += "Morning"` gives `"Good Morning"`.

4. Logical operators: `and`, `or`, `not`.

- `and` — `True` only if both expressions are true: `(a > b) and (b > c)`.
- `or` — `True` if either or both expressions are true.
- `not` — reverses the value of the expression.
- Logical expressions are evaluated in a **short-circuit** manner: evaluation stops as soon as the result is known.
- *Note:* The textbook writes these as `&&`, `||`, `!` (C syntax). In Python they must be written as `and`, `or`, `not`.

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

```
>>> marks = 72
>>> marks >= 40 and marks < 75
True
>>> not (marks >= 40)
False
```

Exam question on this section:

Q. What is an operator? Explain arithmetic, comparison, assignment, and logical operators with examples. **[8 Marks]**

1. **Unary minus:** acts on a single operand and negates its value. If `b = 10`, then `a = -(b)` gives `-10`. (Python has no `++` or `--` operators.)
2. **Bitwise operators** work on the bits of integers (not floats):
   - `&` (AND): bit is 1 only if both bits are 1 — `10 & 12` $= 1010$ `&` $1100 = 1000 = 8$.
   - `|` (OR): bit is 1 if either bit is 1 — `10 | 12` $= 14$.
   - `^` (XOR): bit is 1 if exactly one bit is 1 — `10 ^ 12` $= 6$.
   - `~` (NOT): complements every bit — `~10` $= -11$.
3. **Shift operators:** `<<` shifts bits left (multiplies by 2 for each shift); `>>` shifts bits right (divides by 2). If `x = 0001 1101` (29), `x << 1` gives `0011 1010` (58) and `x >> 1` gives `0000 1110` (14).
4. **Membership operators:** `in` returns `True` if a value is found in a sequence; `not in` returns `True` if it is not found. `'e' in 'hello'` is `True`.
5. **Identity operators:** `is` returns `True` if both operands refer to the same object (`id(a) == id(b)`); `is not` returns `True` otherwise.

```
>>> p = [1, 2]
>>> q = p
>>> r = [1, 2]
>>> p is q, p is r, p == r
(True, False, True)
```

Exam question on this section:

Q. Explain unary, bitwise, shift, membership, and identity operators with examples. **[6 Marks]**

3. Expressions

An **expression** is any legal combination of symbols (variables, constants, and operators) that represents a value. It must have at least one operand and can have one or more operators. Example: `A * B + C - 5`.

Based on the position of the operator:

- **Infix:** operator between operands — `a = b - c` (used in Python).
- **Prefix:** operator before operands — `- b c`.
- **Postfix:** operator after operands — `b c -`.

Based on the data type of the result:

- **Constant expression:** only constants — `8 + 9 - 2`.
- **Integral expression:** integer result — `c = a * b` with integer `a`, `b`.

- **Floating point expression:** float result — `a * b / 2`.
- **Relational expression:** `True` or `False` — `c = a > b`.
- **Logical expression:** combines relational expressions — `a > b and y != 0`.
- **Bitwise expression:** works at bit level — `x = y & z`.
- **Assignment expression:** assigns a value — `c = a + b`.

An illegal expression, such as `a+` or `<y++`, gives a `SyntaxError`.

Exam question on this section:

Q. What is an expression? Explain the types of expressions in Python. **[6 Marks]**

4. Question Bank

Sl.	Question	Marks	Section
1	Explain data types, assignment, multiple assignment, and the Boolean data type.	8	1
2	Explain arithmetic, comparison, assignment, and logical operators with examples.	8	2
3	Explain unary, bitwise, shift, membership, and identity operators.	6	2
4	Differentiate between `/` and `//`, and between `%` and `//`, with examples.	6	2
5	What is an expression? Explain the types of expressions.	6	3

Module 2: Python Fundamentals

Topic 5: Operator Precedence and Type Conversion

Precedence and associativity of operators; type conversion functions; `int()` versus `round()`; type casting and type coercion

1. Operator Precedence and Associativity

When an expression has more than one operator, the **precedence** (priority) of the operators decides the order in which they are evaluated. When operators have the same precedence, their **associativity** decides the order.

Level	Operator(s)	Description
1 (highest)	`( )`	Parentheses
2	`**`	Exponentiation (right to left)
3	`+x`, `-x`, `~x`	Unary plus, minus, complement
4	`*`, `/`, `//`, `%`	Multiply, divide, floor divide, modulo
5	`+`, `-`	Addition, subtraction
6	`<<`, `>>`	Bitwise shifts
7	`&`, then `^`, then `|`	Bitwise AND, XOR, OR
8	`==`, `!=`, `<`, `>`, `<=`, `>=`, `is`, `is not`, `in`, `not in`	Comparison, identity, membership
9	`not`	Logical NOT
10	`and`	Logical AND
11 (lowest)	`or`	Logical OR

Assignment operators (`=`, `+=`, …) are applied last, after the whole right side has been evaluated.

Points to remember:

1. Python follows the order of normal mathematics — **BEDMAS**: Brackets, Exponents, Division and Multiplication, Addition and Subtraction.
2. **Parentheses** change the order: the operation inside them is done first.
3. Operators with the same precedence are evaluated **left to right**, except `**`, which is evaluated **right to left**.

```
>>> 10 + 30 * 5              # * before +
160
>>> (40 + 20) * 30 / 10      # brackets first
180.0
>>> 40 + (20 * 30) / 10
100.0
>>> 100 / 10 / 5             # left to right
2.0
>>> 2 ** 3 ** 2              # right to left: 2 ** 9
512
>>> False == False or True   # == before or
True
>>> False == (False or True)
False
```



Figure 5.1: Precedence (before +) and associativity (** from right to left)*

Note: Some textbook outputs (such as `180` for `(40 + 20) * 30 / 10`) are from Python 2. In Python 3, `/` always gives a float, so the answer is `180.0`.

Exam question on this section:

Q. Explain operator precedence and associativity in Python with examples. **[8 Marks]**

2. Type Conversion

In Python, some operations are not possible between different types of data. For example, `"2" + 4` is not possible because one operand is a string and the other an integer. Type conversion is also needed because `input()` returns a string, which must be converted to a number before calculation.

```
>>> "20" + "30"
'2030'
>>> int("2") + int("3")
5
```

```
x = input("Enter the first number:")        # without conversion
y = input("Enter the second number:")
print(x + y)                                # inputs 6 and 7 -> 67

x = int(input("Enter the first number:"))   # with int() conversion
y = int(input("Enter the second number:"))
print(x + y)                                # inputs 5 and 6 -> 11
```

Type conversion functions:

Function	Description	Example
`int(x)`	Converts x to an integer (truncates floats)	`int(2.90)` → `2`
`float(x)`	Converts x to a floating point number	`float("3.5")` → `3.5`
`str(x)`	Converts x to a string	`str(10)` → `'10'`
`tuple(x)`, `list(x)`, `set(x)`	Converts x to a tuple, list, set	—
`ord(x)`	Converts a single character to its integer value	`ord('A')` → `65`
`chr(x)`	Converts an integer to a character	`chr(66)` → `'B'`
`oct(x)`, `hex(x)`	Converts an integer to an octal / hexadecimal string	`hex(100)` → `'0x64'`
`dict(x)`	Creates a dictionary if x forms key–value pairs	—

(Table 3.8 of the textbook also lists `long()` and `unichr()`; these exist only in Python 2.)

`int()` versus `round()`: `int()` converts a float to an integer by **truncation** (discarding the fractional part), while `round()` rounds to the nearest integer. `round()` can also take the number of decimal places.

```
>>> int(2.90)
2
>>> round(2.90)
3
>>> round(89.567890, 2)
89.57
```

Type casting vs. type coercion:

Sl.	Aspect	Type Casting (Explicit)	Type Coercion (Implicit)
1	Done by	The programmer	Python, automatically
2	How	Using functions such as `int()`, `float()`, `str()`	During evaluation of an expression
3	Example	`int("15") + 3` → `18`	`21 + 2.1` → `23.1`
4	Data loss	Possible (`int(2.9)` → `2`)	None (int is widened to float)

Conclusion: If an argument of the wrong type is passed to a function, Python generates an error (e.g., `int("abc")` gives `ValueError`). Use `help()` to learn which types a function accepts.

Exam question on this section:

Q. What is type conversion? Explain the type conversion functions. Differentiate between type casting and type coercion. **[8 Marks]**

3. Question Bank

Sl.	Question	Marks	Section
1	Explain operator precedence and associativity with examples.	8	1
2	Evaluate: (a) `10 + 30 * 5` (b) `2 ** 3 ** 2` (c) `-19 // 5` (d) `17 % 5` (e) `(40 + 20) * 30 / 10`.	6	1
3	What is type conversion? Explain conversion functions; differentiate type casting and coercion.	8	2
4	Differentiate between `int()` and `round()` with examples.	6	2

Module 2: Python Fundamentals

Topic 6: Basic Debugging Techniques

Debugging; types of errors; preconditions for effective debugging; debugging aids; debugging with the Python IDLE debugger

1. Basic Debugging Techniques

Debugging is a method that involves testing a program during its execution and correcting the code. During debugging, a program is executed several times with different inputs to ensure that it performs its intended task correctly.

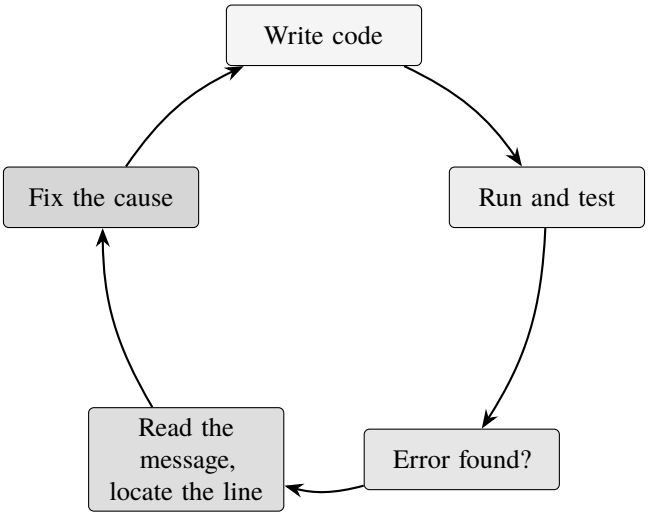


Figure 6.1: The debugging cycle

Types of errors:

Error	Description	Example
Syntax error	The rules of the language are broken; the program does not run	`print("Hello"` — missing bracket
Run-time error (exception)	The program stops while running	`15 / 0` gives `ZeroDivisionError`
Logical error	The program runs but produces a wrong result	`avg = m1 + m2 + m3 / 3`

Preconditions for effective debugging:

1. **Understand the algorithm:** know what the code is supposed to do, so that test cases can be written.
2. **Check correctness:** check the conditions before and after each step.
3. **Code tracing:** trace the execution by hand to understand the logic.
4. **Peer review:** get the code examined by a peer, who may spot errors you missed.

Debugging aids:

1. **Print statements:** insert `print()` to check the value of variables at different points.
2. **Tracebacks:** on a run-time error, Python reports the line number and the cause. The last line gives the error type.
3. **Assert statements:** `assert condition` stops the program with an error message if the condition is `False`.

4. **Debugger:** runs the program line by line with breakpoints, so the programmer can examine variables.

Once an error is detected, its **cause** (not the symptom) should be fixed.

```
m1, m2, m3 = 85, 78, 92
avg = m1 + m2 + m3 / 3              # logical error
print("DEBUG avg =", avg)           # DEBUG avg = 193.66666666666666
avg = (m1 + m2 + m3) / 3            # corrected
assert avg <= 100, "average too big"
print("Average =", avg)             # Average = 85.0
```

Debugging using the Python IDLE debugger:

1. In the shell window, select **Debug** → **Debugger**. The **Debug Control** window opens and `[DEBUG ON]` appears.
2. In the program window, right-click a line and choose **Set Breakpoint**; the line turns yellow.
3. Press **F5** to run. Use **Go** (run to the next breakpoint), **Step** (next line, entering functions), **Over** (next line, without entering functions), **Out** (finish the current function), and **Quit** (stop).
4. The **Locals** pane shows the values and types of variables as they change.

Frequent errors: wrongly spelt variable names, changing the case of variable names (this creates a new variable), and failing to initialize a variable.

Conclusion: Testing shows that bugs exist; debugging locates and removes them.

Exam question on this section:

> **Q.** What is debugging? Explain the types of errors, debugging aids, and debugging using the Python IDLE debugger. **[8 Marks]**

2. Question Bank

Sl.	Question	Marks	Section
1	What is debugging? Explain types of errors, debugging aids, and the IDLE debugger.	8	1
2	Differentiate between syntax errors, run-time errors, and logical errors with an example of each.	6	1
3	What are the preconditions for effective debugging? Explain print statements, tracebacks, and assert statements as debugging aids.	6	1

Module 2: Python Fundamentals

Topic 7: Laboratory Programs

Module 2 laboratory component — four programs on input/output, data types, operators and type conversion (all programs are in Python 3, with verified output)

1. Program 1: Student Details, Total, and Average

```
name = input("Enter student name: ")
usn = input("Enter USN: ")
m1 = float(input("Enter marks in course 1: "))
m2 = float(input("Enter marks in course 2: "))
m3 = float(input("Enter marks in course 3: "))

total = m1 + m2 + m3
average = total / 3

print("\n------- Student Details -------")
print("Name    :", name)
print("USN     :", usn)
print("Marks   :", m1, m2, m3)
print("Total   :", total)
print("Average :", format(average, ".2f"))
```

Output

```
Enter student name: Ananya K
Enter USN: 4MT26CS001
Enter marks in course 1: 85
Enter marks in course 2: 78
Enter marks in course 3: 92

------- Student Details -------
Name    : Ananya K
USN     : 4MT26CS001
Marks   : 85.0 78.0 92.0
Total   : 255.0
Average : 85.00
```

Laboratory question:

Q. Write a Python program to read the Student Name, USN, and marks of three courses, calculate the total and average marks, and display the student details in a formatted way. **[6 Marks]**

2. Program 2: Area and Circumference of a Circle

```
radius = float(input("Enter the radius of the circle: "))
PI = 3.14159

area = PI * radius * radius
circumference = 2 * PI * radius

print("Area of the circle          =", format(area, ".2f"))
print("Circumference of the circle =", format(circumference, ".2f"))
print("Area as an integer          =", int(area))
```

Output

```
Enter the radius of the circle: 7
Area of the circle          = 153.94
Circumference of the circle = 43.98
Area as an integer          = 153
```

Laboratory question:

Q. Write a Python program to read the radius of a circle, calculate and display the area and circumference, and convert the area into an integer. **[6 Marks]**

3. Program 3: Values and Data Types

```
i = int(input("Enter an integer: "))
f = float(input("Enter a floating-point number: "))
s = input("Enter a string: ")
b = input("Enter a Boolean (True/False): ") == "True"

print("\nValue :", i, " Type :", type(i))
print("Value :", f, " Type :", type(f))
print("Value :", s, " Type :", type(s))
print("Value :", b, " Type :", type(b))

print("\nInteger + Float   =", i + f)
print("Integer * 2       =", i * 2)
print("Float / 2         =", f / 2)
print("String + ' MITE'  =", s + " MITE")
print("String * 3        =", s * 3)
print("Length of string  =", len(s))
print("Boolean + 1       =", b + 1)
```

Output

```
Enter an integer: 25
Enter a floating-point number: 3.5
Enter a string: Python
Enter a Boolean (True/False): True

Value : 25  Type : <class 'int'>
Value : 3.5  Type : <class 'float'>
Value : Python  Type : <class 'str'>
Value : True  Type : <class 'bool'>

Integer + Float   = 28.5
Integer * 2       = 50
Float / 2         = 1.75
String + ' MITE'  = Python MITE
String * 3        = PythonPythonPython
Length of string  = 6
Boolean + 1       = 2
```

Note: `bool(input())` is not used, because `bool()` of any non-empty string (even `"False"`) is `True`. Comparing the input with `"True"` gives a correct Boolean value.

Laboratory question:

Q. Write a Python program that accepts an integer, a floating-point number, a string, and a Boolean value, displays their values and data types, and performs suitable arithmetic and string operations. **[6 Marks]**

4. Program 4: Arithmetic Operations on Two Numbers

```
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))

print("Addition       :", a, "+", b, "=", a + b)
print("Multiplication :", a, "*", b, "=", a * b)
print("Division       :", a, "/", b, "=", a / b)
print("Floor Division :", a, "//", b, "=", a // b)
print("Modulus        :", a, "%", b, "=", a % b)
print("Exponentiation :", a, "**", b, "=", a ** b)
```

Output

```
Enter the first number: 17
Enter the second number: 5
Addition       : 17 + 5 = 22
Multiplication : 17 * 5 = 85
Division       : 17 / 5 = 3.4
Floor Division : 17 // 5 = 3
Modulus        : 17 % 5 = 2
Exponentiation : 17 ** 5 = 1419857
```

Laboratory question:

Q. Write a Python program that accepts two numbers and performs addition, multiplication, division, floor division, modulus, and exponentiation, and displays the results. **[6 Marks]**