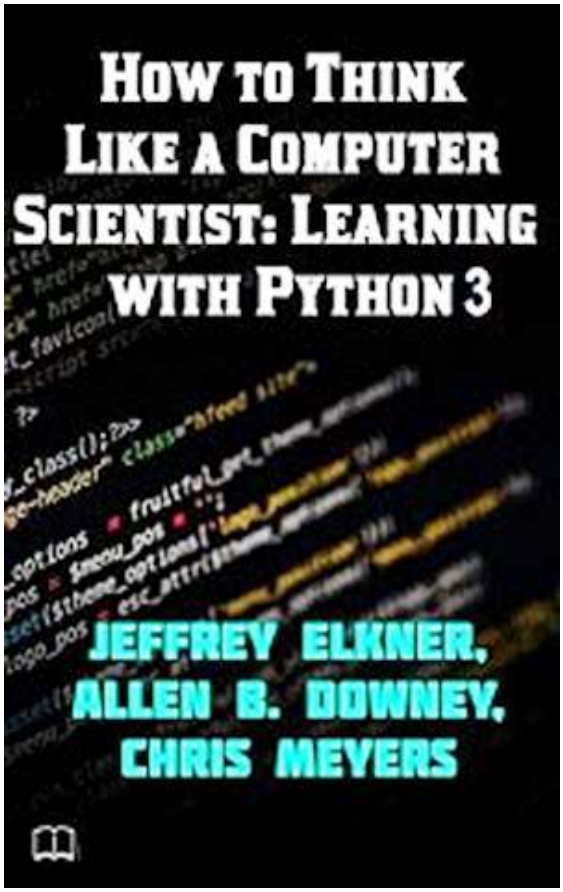
AMC Engineering College

Bangalore – 560083.

1BPLC105B/205B PYTHON PROGRAMMING

SEMESTER 2 – ACADEMIC YEAR 2025-2026



LECTURE NOTES

Prof. Parthasarathy P V

Department of Computer Science and Engineering

url: www.thagadoor.in/vtu/python/

1BPLC105B/205B PYTHON PROGRAMMING

Module-1

The way of the program *(Pg 3– 7):* The Python programming language, what is a program? What is debugging? Syntax errors, Runtime errors, Semantic errors, Experimental debugging.

Variables, Expressions and Statements *(Pg 11– 21):* Values and data types, Variables, Variable names and keywords, Statements, Evaluating expressions, Operators and operands, Type converter functions, Order of operations, Operations on strings, Input, Composition, The modulus operator.

Iteration *(Pg 29– 30):* Assignment, Updating variables, the for loop, the while statement, The Collatz 3n + 1 sequence, tables, two-dimensional tables, break statement, continue statement, paired data, Nested Loops for Nested Data.

Functions *(Pg 39– 47):* Functions with arguments and return values.

Chapters: 1.1-1.7, 2.1-2.12, 3.3, 4.4, 4.5

The way of the program (Pg 3– 7)

The Python programming language, What is a program? What is debugging? Syntax errors, Runtime errors, Semantic errors, Experimental debugging.

The Python programming language

- ***Python*** is an example of a ***high-level language***; other high-level languages you might have heard of are ***C++, PHP, Pascal, C#,*** and ***Java***.
- There are also ***low-level languages***, sometimes ***referred*** to as ***machine languages*** or ***assembly languages.***

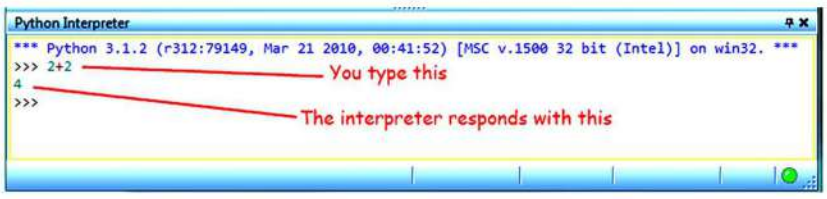
High-level languages – Advantages

- **First**, it is much ***easier to program*** in a high-level language so
  - programs take less time to write
  - they are shorter and easier to read, and
  - they are more likely to be correct.
- **Second**, high-level languages are ***portable***, meaning that they can run on different kinds of computers with few or no modifications.

Python Interpreter

- The engine that ***translates*** and ***runs Python*** is called the ***Python Interpreter***: There are ***two*** ways to use it:
  1. immediate mode and
  2. script mode.

1. In ***immediate*** mode, you type Python expressions into the Python Interpreter window, and the interpreter immediately shows the result:



- The **>>>** is called the **Python prompt**.

- The interpreter uses the prompt to indicate that it is ready for instructions. We typed 2 + 2, and the interpreter evaluated our expression, and replied 4, and on the next line it gave a new prompt, indicating that it is ready for more input.

2. In ***script*** mode: you can ***write a program in a file and use the interpreter to execute the contents*** of the file. Such a file is called a ***script***. ***Scripts*** have the advantage that they can be saved to disk, printed, and so on.

- Working directly in the interpreter is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems. Anything longer than a few lines should be put into a script.
- When you are writing a script you need something like a text editor. A few examples of text editors are **Notepad, Notepad++, vim, emacs** and **sublime**.
- For Python, there are programs that include both a text editor and a way to in teract with the interpreter. We call these development environments (sometimes Integrated Development Environment or IDE). For Python these can include
  - Spyder
  - Thonny or
  - IDLE.
- There are also development environments that run in your browser. One example of this is **Jupyter** Notebook.

The important thing to remember: Python itself does not care in what editor you write your code. As long as you write correct syntax (with the right tabs and spaces) Python can run your program. The editor is only there to help you.

What is a program?

- A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be
  - something mathematical, such as
    - solving a system of equations or
    - finding the roots of a polynomial
  - it can also be a symbolic computation, such as
    - searching and replacing text in a document or (strangely enough)
    - compiling a program.
- few basic instructions appear in just about every language.
- **Input:** Get data from the keyboard, a file, or some other device.
- **Output:** Display data on the screen or send data to a file or other device.
- **Math:** Perform basic mathematical operations like addition and multiplication.
- **Conditional execution:** Check for certain conditions and execute the appropriate sequence of statements.
- **Repetition**: Perform some action repeatedly, usually with some variation.

What is debugging?

- Programming is a complex process, and because it is done by human beings, it often leads to errors.
- Programming errors are called bugs and the process of tracking them down and correcting them is called **debugging**. Use of the term bug to describe small engineering difficulties dates back to at least 1889, when Thomas Edison had a bug with his phonograph.

- Three kinds of errors can occur in a program:
  - syntax errors
  - runtime errors, and
  - semantic errors.
- It is useful to distinguish between them in order to track them down more quickly.

Syntax errors

- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message.
- **Syntax** refers to the structure of a program and the rules about that structure.
- For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**.
- If there is a single syntax error anywhere in your program, Python will display an error message and quit, and you will not be able to run your program.

Runtime errors

- The second type of error is a **runtime error**, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.
- Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic errors

- The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.
- The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Experimental debugging

- One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again.
- If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one.
- As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth.
- For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want.

- The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.

Formal and natural languages

- Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.
- Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use formal language to represent the chemical structure of molecules.
- And most importantly: ***Programming languages are formal languages that have been designed to express computations. Formal languages tend to have strict rules about syntax.***

The first program

- Traditionally, the first program written in a new language is called Hello, World!

Program Code	Output
print("Hello, World!")	Hello, World!
print(6 + 4 * 9)	42

- This is an example of using the print function, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result shown is **Hello, World!**
- The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Comments

- As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing.
- A **comment** in a computer program is text that is intended only for the human reader — it is completely ignored by the interpreter. In Python, the # token starts a comment. The rest of the line is ignored. Here is a new version of Hello, World!.

```
#-----------------------------------------------
# This demo program shows off how elegant Python is!
#-----------------------------------------------
print("Hello, World!")      # Isn't this easy!
```

Glossary

algorithm	*A set of specific steps for solving a category of problems.*
bug	*An error in a program.*
comment	*Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.*
debugging	*The process of finding and removing any of the three kinds of programming errors.*

exception	Another name for a runtime error.
formal language	Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.
high-level language	A programming language like Python that is designed to be easy for humans to read and write.
immediate mode	A style of using Python where we type expressions at the command prompt, and the results are shown immediately. Contrast with script, and see the entry under Python shell.
interpreter	The engine that executes your Python scripts or expressions.
low-level language	A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.
natural language	Any one of the languages that people speak that evolved naturally.
object code	The output of the compiler after it translates the program.
parse	To examine a program and analyze the syntactic structure.
portability	A property of a program that can run on more than one kind of computer.
print function	A function used in a program or script that causes the Python interpreter to display a value on its output device.
problem solving	The process of formulating a problem, finding a solution, and expressing the solution.
program	a sequence of instructions that specifies to a computer actions and computations to be performed.
Python shell	An interactive user interface to the Python interpreter. The user of a Python shell types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter for processing. The word shell comes from Unix. In the PyScripter used in this RLE version of the book, the Interpreter Window is where we'd do the immediate mode interaction.
runtime error	An error that does not occur until the program has started to execute but that prevents the program from continuing.
script	A program stored in a file (usually one that will be interpreted).
semantic error	An error in a program that makes it do something other than what the programmer intended.
semantics	The meaning of a program.
source code	A program in a high-level language before being compiled.
Syntax	The structure of a program.
syntax error	An error in a program that makes it impossible to parse — and therefore impossible to interpret.
token	One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

==========

Variables, Expressions and Statements (Pg 11– 21)

Values and data types, Variables, Variable names and keywords, Statements, Evaluating expressions, Operators and operands, Type converter functions, Order of operations, Operations on strings, Input, Composition, The modulus operator.

==========

- **Value** is one of the **fundamental things** — like a **letter** or a **number** — **that a program manipulates.** The values we have seen so far are 4 (the result when we added 2 + 2), and "Hello, World!".
- These **values** are **classified into different classes**, or **data types**: 4 is an integer, and "Hello, World!" is a string, so-called because it contains a string of letters.
- If you are not sure what class a value falls into, Python has a function called type which can tell you.

```
>>> type("Hello, World!")
<class 'str'>
>>> type(17)
<class 'int'>
>>> type(3.2)
<class 'float'>
>>> type("17")
<class 'str'>
>>> type("3.2")
<class 'str'>
```

- **Strings** belong to the class **str** and **integers** belong to the class **int**. Less obviously, numbers with **a decimal point** belong to a class called **float**, because these numbers are represented in a format called **floating-point**.
- What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

- Strings in Python can be enclosed in either single quotes (') or double quotes ("), or three of each (''' or """)

```
>>> type('This is a string.')
<class 'str'>
>>> type("And so is this.")
<class 'str'>
>>> type("""and this.""")
<class 'str'>
>>> type('''and even this...''')
<class 'str'>
```

- Double quoted strings can contain single quotes inside them, as in "**Bruce's beard**".
- Single quoted strings can have double quotes inside them, as in '**The knights who say "Ni!"**'.
- Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes. Triple quoted strings can even span multiple lines.

```
>>> print('''"Oh no", she exclaimed, "Ben's bike is broken!"''')
"Oh no", she exclaimed, "Ben's bike is broken!"
>>>
>>> message = """This message will
... span several
... lines."""
>>> print(message)
This message will
span several
lines.
>>>
```

- Python doesn't care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings: once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value. But when the interpreter wants to display a string, it has to decide which quotes to use to make it look like a string.

```
>>> 'This is a string.'
'This is a string.'
>>> """And so is this."""
'And so is this.'
```

- So the Python language designers usually choose to surround their strings by single quotes.
- When you type a large integer, you might be tempted to use commas between groups of three digits, as in 42,000. This is not a legal integer in Python, but it does mean something else, which is legal:

```
>>> 42000
42000
>>> 42,000
(42, 0)
```

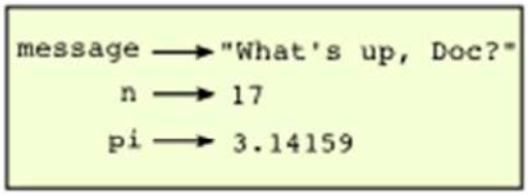
- Well, that's not what we expected at all! Because of the comma, Python chose to treat this as a pair of values. Remember not to put commas or spaces in your integers, no matter how big they are.

Variables

- One of the most powerful features of a programming language is the ability to manipulate variables.
- A **variable** is a name that **refers to a value**. The **assignment statement** gives a **value to a variable**:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```


- This example makes three assignments.
  - The first assigns the string value "What's up, Doc?" to a variable named message.
  - The second gives the integer 17 to n, and
  - The third assigns the floating-point number 3.14159 to a variable called pi.
- A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a state snapshot because it shows what state each of the variables is in at a particular instant in time.
- This diagram shows the result of executing the assignment statements:



- If you ask the interpreter to evaluate a variable, it will produce the value that is currently linked to the variable:

```
>>> message
'What's up, Doc?'
>>> n
17
>>> pi
3.14159
```

Variable names and keywords

- Variable names can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. **Bruce** and **bruce** are different variables.
- The underscore character **(_)** can appear in a name. It is often used in names with multiple words, such as **my_name** or **price_of_tea_in_china.**
- There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.
- If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

- **76trombones** is illegal **because** it does not begin with a letter.
- **more$** is illegal because it contains an illegal character, the dollar sign.
- But what's wrong with class? **class** is one of the Python keywords.

- **Keywords** define the language's syntax rules and structure, and they **cannot be used as variable names**. Python has 34 keywords.

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

- Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for.

Statements

- A **statement** is an **instruction** that the **Python interpreter can execute**. We have only seen the assignment statement so far. Some other kinds of statements are **while** statements, **for** statements, **if** statements, and **import** statements.
- ***When you type a statement on the command line, Python executes it. Statements don't produce any result.***

Evaluating expressions

- An **expression** is a **combination of values, variables, operators**, and **calls to functions**. If you type an expression at the Python prompt, the interpreter evaluates it and displays the result:

```
>>> 1 + 1
2
>>> len("hello")
5
```

- In this example **len** is a **built-in Python function** that **returns the number of characters in a string**.
- The **evaluation of an expression produces a value**, which is why expressions can appear on the right-hand side of assignment statements.

- A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> y = 3.14
>>> x = len("hello")
>>> x
5
>>> y
3.14
```

Operators and operands

- **Operators** are **special tokens** that **represent computations** like **addition**, **multiplication** and **division**. The values the **operator** uses are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

```
20+32    hour-1    hour*60+minute    minute/60    5**2    (5+9)*(15-7)
```

The tokens +, −, and ∗, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (∗) is the token for multiplication, and ∗∗ is the token for exponentiation.

```
>>> 2 ** 3
8
>>> 3 ** 2
9
```

- When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.
- Addition, subtraction, multiplication, and exponentiation all do what you expect.
- Example: so let us convert 645 minutes into hours:

```
>>> minutes = 645
>>> hours = minutes / 60
>>> hours
10.75
```

- In Python 3, the division operator **/** always yields a **floating point** result. What we might have wanted to know was how many whole hours there are, and how many minutes remain.
- Python gives us **two different** flavors of the **division operator**.
- The **second**, called floor division uses the token **//**. Its result is always a whole number — and if it has to adjust the number it always moves it to the left on the number line. So 6 **//** 4 yields 1, but-6 **//** 4 might surprise you!

```
>>> 7 / 4
1.75
>>> 7 // 4
1
>>> minutes = 645
>>> hours = minutes // 60
>>> hours
10
```

Type converter functions

- Python functions, int, float and str, which will (attempt to) convert their arguments into types int, float and str respectively. We call these type converter functions.
- The int function can take a floating point number or a string, and turn it into an int. For f loating point numbers, it discards the decimal portion of the number — a process we call truncation towards zero on the number line. Let us see this in action:

```
>>> int(3.14)
3
>>> int(3.9999)  # This doesn't round to the closest int!
3
>>> int(3.0)
3
>>> int(-3.999)  # Note that the result is closer to zero
-3
>>> int(minutes / 60)
10
>>> int("2345") # Parse a string to produce an int
2345
>>> int(17)     # It even works if arg is already an int
17
>>> int("23 bottles")
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23 bottles'
```

- The type converter ***float*** can turn an ***integer***, a float, or a syntactically legal string into a float:

  >>> float(17)

  17.0

  >>> float("123.45")

  123.45

- The type converter ***"str"*** turns its argument into a string:

  >>> str(17)

  '17'

  >>> str(123.45)

  '123.45'

Order of operations

- When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:
  1. **P**arentheses have the highest precedence and can be used to force an expression to eval uate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1)is4, and (1+1)**(5-2)is8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.
  2. **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.
  3. **M**ultiplication and both **D**ivision operators have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 5-2*2 is 1, not 6.
  4. Operators with the same precedence are **evaluated from left-to-right**. In algebra we say they are left-associative. So in the expression **6-3+2**, the subtraction happens first, yielding 3. We then add 2 to get the **result 5**.

```
>>> 2 ** 3 ** 2      # The right-most ** operator gets done first!
512
>>> (2 ** 3) ** 2    # Use parentheses to force the order you want!
64
```

Operations on strings

- In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```
>>> message - 1          # Error
>>> "Hello" / 123        # Error
>>> message * "Hello"    # Error
>>> "15" + 2             # Error
```

- Interestingly, the + operator does work with strings, but for strings, the + operator represents concatenation, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

```
fruit = "banana"
baked_good = " nut bread"
print(fruit + baked_good)
```

- The ***output*** of this program is ***"banana nut bread".*** The space before the word ***nut*** is part of the string, and is necessary to produce the space between the concatenated strings.
- The ***** operator also works on strings; it performs repetition. For example, ***'Fun'*3*** is ***'FunFunFun'***. One of the operands has to be a string; the other has to be an integer.
- On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect "Fun"*3 to be the same as "Fun"+"Fun"+"Fun", and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication.

Input

- There is a built-in function in Python for getting input from the user:

  n = input("Please enter your name: ")

- The user of the program can enter the name and click OK, and when this happens the text that has been entered is returned from the input function, and in this case assigned to the variable name.
- Even if you asked the user to enter their age, you would get back a string like "17". It would be your job, as the programmer, to convert that string into a int or a float, using the int or float converter functions.

Composition

- One of the most useful features of programming languages is their ability to take small building blocks and compose them into larger chunks.
- For example, we know how to get the user to enter some input, we know how to convert the string we get into a float, we know how to write a complex expression, and we know how to print values.
- Let's put these together in a small four-step program that asks the user to input a value for the radius of a circle, and then computes the area of the circle from the formula **Area=**$A = \pi r^2$

Firstly, we'll do the four steps one at a time:

```
response = input("What is your radius? ")
r = float(response)
area = 3.14159 * r**2
print("The area is ", area)
```

Now let's compose the first two lines into a single line of code, and compose the second two lines into another line of code.

```
r = float( input("What is your radius? ") )
print("The area is ", 3.14159 * r**2)
```

If we really wanted to be tricky, we could write it all in one statement:

```
print("The area is ", 3.14159*float(input("What is your radius?"))**2)
```

- Such compact code may not be most understandable for humans, but it does illustrate how we can compose bigger chunks from our building blocks.
- If you're ever in doubt about whether to compose code or fragment it into smaller steps, try to make it as simple as you can for the human to follow. My choice would be the first case above, with four separate steps.

The modulus operator (%)

- The modulus operator works on integers (and integer expressions) and gives the remainder when the first number is divided by the second.
- In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators. It has the same precedence as the multiplication operator.

```
>>> q = 7 // 3 #This is integer division operator
>>> print(q)
2
>>> r  = 7 % 3
>>> print(r)
1
```

So 7 divided by 3 is 2 with a remainder of 1.

- The modulus operator turns out to be surprisingly useful.
- For example, you can ***check whether one number is divisible by another***—if **x % y** is zero, then **x is divisible by y**.
- Also, you can ***extract the right-most digit or digits from a number***. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.
- It is also extremely useful for doing conversions, say from seconds, to hours, minutes and seconds. So let's write a program to ask the user to enter some seconds, and we'll convert them into hours, minutes, and remaining seconds.

```
total_secs = int(input("How many seconds, in total? "))
hours = total_secs // 3600
secs_still_remaining = total_secs % 3600
minutes = secs_still_remaining // 60
secs_finally_remaining = secs_still_remaining % 60
print("Hrs=", hours, " mins=",minutes,"secs=", secs_finally_remaining)
```

Output

```
How many seconds, in total? 4000
Hrs= 1  mins= 6 secs= 40
```

Ch 3.3. Iteration (Pg 85– 105)

Assignment, Updating variables, the for loop, the while statement, The Collatz 3n + 1 sequence, tables, two-dimensional tables, break statement, continue statement, paired data, Nested Loops for Nested Data.

==========

Assignment

- It is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
airtime_remaining = 15
print(airtime_remaining)
airtime_remaining = 7
print(airtime_remaining)
```

The output of this program is:

```
15
7
```

- Because the ***first-time airtime_remaining*** is printed, its value is ***15***, and the ***second time***, its value is ***7***.
- It is especially important to distinguish between an assignment statement and a Boolean expression that tests for equality. Because Python uses the equal token (=) for assignment, it is tempting to interpret a statement like a = b as a Boolean test. Unlike mathematics, it is not!
- Remember that the ***Python token for the equality operator is ==.***
- Note too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a. But in ***Python, the statement a = 7 is legal and 7 = a is not.***
- In Python, an assignment statement can make two variables equal, but because further assignments can change either of them, they don't have to stay that way:

```
a = 5
b = a     # After executing this line, a and b are now equal
a = 3     # After executing this line, a and b are no longer equal
```

- The third line changes the value of a but does not change the value of b, so they are no longer equal.

Updating variables

- When an assignment statement is executed, the right-hand side expression (i.e. the expression that comes after the assignment token) is evaluated first. This produces a value. Then the assignment is made, so that the variable on the left-hand side now refers to the new value.
- One of the most common forms of assignment is an update, where the new value of the variable depends on its old value. Deduct 40 cents from my airtime balance, or add one run to the scoreboard.

```
n = 5
n = 3 * n + 1
```

- Line 2 means get the current value of n, multiply it by three and add one, and assign the answer to n, thus making n refer to the value. So after executing the two lines above, n will point/refer to the integer 16.
- If you try to get the value of a variable that has never been assigned to, you'll get an error:

```
>>> w = x + 1
Traceback (most recent call last):
  File "<interactive input>", line 1, in
NameError: name 'x' is not defined
```

- Before you can update a variable, you have to initialize it to some starting value, usually with a simple assignment:

  runs_scored = 0

  ...

  runs_scored = runs_scored + 1

- Line 3 —updating a variable by adding 1 to it —is very common. Itis called an increment of the variable; subtracting 1 is called a decrement.
- Sometimes programmers also talk about bumping a variable, which means the same as incrementing it by 1. This is commonly done with the += operator.

  runs_scored = 0

  ...

  runs_scored += 1

The "for" loop

- When we drew the square, it was quite tedious. We had to explicitly repeat the steps of moving and turning four times. If we were drawing a hexagon, or an octagon, or a polygon with 42 sides, it would have been worse.
- So, a basic building block of all programs is to be able to repeat some code, over and over again.
- Python's for loop solves this for us. Let's say we have some friends, and we'd like to send them each an email inviting them to our party. We don't quite know how to send email yet, so for the moment we'll just print a message for each friend:

```
for f in ["Joe","Zoe","Brad"]:
    invite = "Hi " + f + ".  Please come to my party on Saturday!"
    print(invite)
```

When we run this, the output looks like this:

```
Hi Joe.  Please come to my party on Saturday!
Hi Zoe.  Please come to my party on Saturday!
Hi Brad.  Please come to my party on Saturday!
```

- The variable ***"f"*** in the ***"for"*** statement at line 1 is ***called the loop variable***. We could have chosen any other variable name instead. ***Running through all the items in a list is called traversing the list,*** or ***traversal.***
- Lines 2 and 3 are the loop body. The loop body is always indented. The indentation determines exactly what statements are "in the body of the loop".
- On each iteration or pass of the loop, first a check is done to see if there are still more items to be processed.
- If there are none left (this is called the terminating condition of the loop), the loop has finished. Program execution continues at the next statement after the loop body, (e.g. in this case the next statement below the comment in line 4).

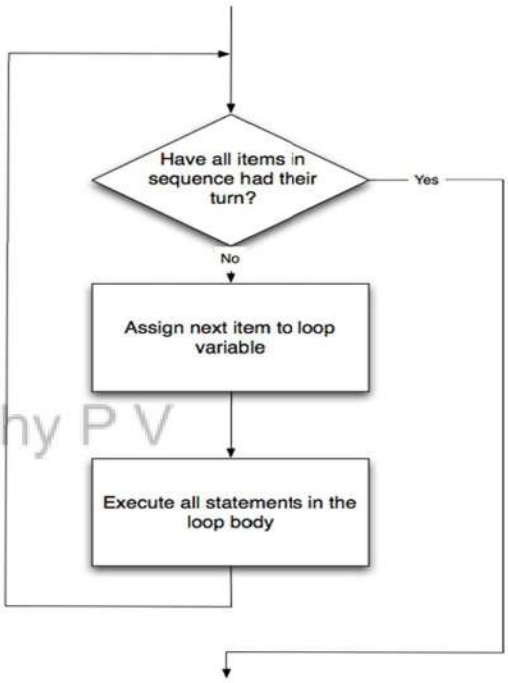
- If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 3 times, and each time ***"f"*** will refer to a different friend.
- At the end of each execution of the body of the loop, Python returns to the ***"for"*** statement, to see if there are more items to be handled, and to assign the next one to ***"f"***.

Flow of Execution of the *"for"* loop

- As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the control flow, of the flow of execution of the program. When humans execute programs, they often use their finger to point to each statement in turn. So we could think of control flow as "Python's moving finger".
- Control flow until now has been strictly top to bottom, one statement at a time. The ***"for"*** loop changes this.

Flowchart of a *"for"* loop

- Control flow is often easy to visualize and understand if we draw a flowchart. This shows the exact steps and logic of how the ***"for"*** statement executes.
- **Let us write some code now to sum up all the elements in a list of numbers.**
- *Do this by hand first, and try to isolate exactly what steps you take. You'll find you need to keep some "running total" of the sum so far, either on a piece of paper, in your head, or in your calculator. Remembering things from one step to the next is precisely why we have variables in a program: so we'll need some variable to remember the "running total". It should be initialized with a value of zero, and then we need to traverse the items in the list. For each item, we'll want to update the running total by adding the next number to it.*




```
numbers = [5, 6, 32, 21, 9]
running_total = 0
for number in numbers:
    running_total = running_total + number
print(running_total)
```

Output: 73

The "while" statement

- Here is a fragment of code that demonstrates the use of the **"while"** statement:

General Form (Syntax)	Example	Output

`while <CONDITION>:`
`  <STATEMENT>`
	`n = 6`
	`current_sum = 0`
	`i = 0`
	`while i <= n:`
	`  current_sum += i`
	`  i += 1`
	`print(current_sum)`
		21

- Here is precise flow of execution for a **"while"** statement:
  - Evaluate the condition at line 5, yielding a value which is either False or True.
  - If the value is False, exit the while statement and continue execution at the next statement (line 7 in this case).
  - If the value is True, execute each of the statements in the body (lines 5 and 6) and then go back to the while statement at line 4.
- The body consists of all of the statements indented below the while keyword.
- Notice that if the loop condition is False the first time we get loop, the statements in the body of the loop are never executed.
- The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop.**
- In the case here, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of i increments each time through the loop, so eventually it will have to exceed n. In other cases it is not so easy, maybe even impossible, to tell if the loop will ever terminate.
- What you will notice here is that the while loop is more work for you — the programmer — than the equivalent for loop. When using a while loop one has to manage the loop variable yourself: give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.
- By comparison, here is an equivalent snippet that uses for instead:

```
n = 6
current_sum = 0
for i in range(n+1):
    current_sum += i
print(current_sum)
```


The Collatz 3n + 1 sequence

- The "computational rule" for creating the sequence is to start from some given n, and to generate the next term of the sequence from n, either by halving n, (whenever n is even), or else by multiplying it by three and adding 1. The sequence terminates when n reaches 1.
- This Python snippet captures that algorithm:

```
n = 10
while n != 1:
    print(n, end=", ")
    if n % 2 == 0:
        n = n // 2
    else:
        n = n * 3 + 1
print(n, end=".\n")
```

Output

```
10, 5, 16, 8, 4, 2, 1.
```

- Notice first that the print function on line 3 has an extra argument end=", ". This tells the print function to follow the printed string with whatever the programmer chooses (in this case, a comma followed by a space), instead of ending the line. So each time something is printed in the loop, it is printed on the same output line, with the numbers separated by commas.
- The call to print(n, end=".\n") at line 8 after the loop terminates will then print the final value of n followed by a period and a newline character.
- The condition for continuing with this loop is n != 1, so the loop will continue running until it reaches its termination condition, (i.e. n == 1).
- Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by 2 using integer division. If it is odd, the value is replaced by n * 3 + 1.
- See if you can find a small starting number that needs more than a hundred steps before it terminates.
- A German mathematician called Lothar Collatz: the Collatz conjecture (also known as the 3n + 1 conjecture), is that this sequence terminates for all positive values of n. So far, no one has been able to prove it or disprove it! (A conjecture is a statement that might be true, but nobody knows for sure.)
- Think carefully about what would be needed for a proof or disproof of the conjecture "All positive integers will eventually converge to 1 using the Collatz rules". With fast computers we have been able to test every integer up to very large values, and so far, they have all eventually ended up at 1. But who knows? Perhaps there is some as-yet untested number which does not reduce to 1.
- You'll notice that if you don't stop when you reach 1, the sequence gets into its own cyclic loop: 1, 4, 2, 1, 4, 2, 1, 4 . . . So one possibility is that there might be other cycles that we just haven't found yet.

Wikipedia has an informative article about the Collatz conjecture. The sequence also goes under other names (Hail stone sequence, Wonderous numbers, etc.), and you'll find out just how many integers have already been tested by computer, and found to converge!

https://en.wikipedia.org/wiki/Collatz_conjecture

Choosing between "for - definite iteration" and "while - indefinite iteration"

- Use a **"for"** loop, before you start looping, the maximum number of times that you'll need to execute the body. For example, if you're traversing a list of elements, you know that the maximum number of loop iterations you can possibly need is "all the elements in the list". Or if you need to print the 12 times table, we know right away how many times the loop will need to run. So any problem like "iterate this weather model for 1000 cycles", or "search this list of words", "find all prime numbers up to 10000" suggest that a for loop is best.
- **We call this case "definite iteration"** — we know ahead of time some definite bounds for what is needed.

- By contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when (of if) this will happen, as we did in this 3n + 1 problem, you'll need a **"while"** loop.
- This case is called "**indefinite iteration**" — we're not sure how many iterations we'll need — we cannot even establish an upper bound!

Tables

- One of the things loops are good for is generating tables. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.
- When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.
- For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation.
- In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.
- Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
for x in range(6):
    print(x, "\t", 2**x)
```

```
Output
0   1
1   2
2   4
3   8
4   16
5   32
```

- The string "\t" represents a tab character. The backslash character in "\t" indicates the beginning of an escape sequence. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a newline.
- An escape sequence can appear anywhere in a string; in this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?
- As characters and strings are displayed on the screen, an invisible marker called the cursor keeps track of where the next character will go.
- After a print function, the cursor normally goes to the beginning of the next line.
- The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up.

Two-dimensional tables

- A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. Let's say you want to print a multiplication table for the values from 1 to 6.
- A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
for i in range(1, 7):
    print(2 * i, end="   ")
print()
```

output of the program is:

```
2   4   6   8   10   12
```

- Here we've used the range function, but made it start its sequence at 1. As the loop executes, the value of i changes from 1 to 6. When all the elements of the range have been assigned to i, the loop terminates. Each time through the loop, it displays the value of 2 * i, followed by three spaces.
- Again, the extra end=" " argument in the print function suppresses the newline, and uses three spaces instead. After the loop completes, the call to print at line 3 finishes the current line, and starts a new line.

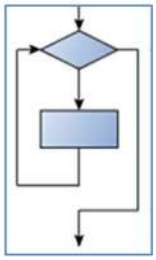
Break statement

- The break statement is used to immediately leave the body of its loop. The next statement to be executed is the first one after the body:

```
for i in [12, 16, 17, 24, 29]:
    if i % 2 == 1: # If the number is odd
        break      # ...immediately exit the loop
    print(i)
print("done")
```

This prints:

```
12
16
done
```



The pre-test loop — standard loop behaviour

- *"for"* and *"while"* loops do their tests at the start, before executing any part of the body. They're called pre-test loops, because the test happens before (pre) the body. break and return (discussed later) are our tools for adapting this standard behaviour.

Continue statement

- This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, for the current iteration. But the loop still carries on running for its remaining iterations.

```
for i in [12, 16, 17, 24, 29, 30]:
    if i % 2 == 1: # If the number is odd
        continue   # Don't process it
    print(i)
print("done")
```

prints:

```
12
16
24
30
done
```

Paired data

- We've already seen lists of names and lists of numbers in Python. We're going to peek ahead in the textbook a little, and show a more advanced way of representing our data. Making a pair of things in Python is as simple as putting them into parentheses, like this:

```
year_born = ("Prof. Parthasarathy PV", 1971)
```

- We can put many pairs into a list of pairs:

```
amc2025 = [("Sonu", 2009), ("Suhruth", 2007), ("Supriya", 2008), ("Tanuja", 2009)]
```

- Here is a quick sample of things we can do with structured data like this. First, print all the "*amc2025*":

Python Code	Output
print(*amc2025*) print(len(*amc2025*))	[("Sonu", 2009), ("Suhruth", 2007), ("Supriya", 2008), ("Tanuja", 2009)] 4

- Notice that the ***amc2025*** list has just 4 elements, each of them pairs.
- Now we print the names of those ***amc2025ians*** born before on ***2009***:

Python Code	Output
for name, year in amc2025: *if year == 2009:* *print(name)*	*Sonu* *Tanuja*

- This demonstrates something we have not seen yet in the for loop: instead of using a single loop control variable, we've used a pair of variable names, (name, year), instead. The loop is executed three times—once for each pair in the list, and on each iteration both the variables are assigned values from the pair of data that is being handled.

Nested Loops for Nested Data

Now we'll come up with an even more adventurous list of structured data. In this case, we have a list of students. Each student has a name which is paired up with another list of subjects that they are enrolled for:

```
students = [
    ("John", ["CompSci", "Physics"]),
    ("Vusi", ["Maths", "CompSci", "Stats"]),
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```


Here we've assigned a list of five elements to the variable `students`. Let's print out each student name, and the number of subjects they are enrolled for:

```
# Print all students with a count of their courses.
for name, subjects in students:
    print(name, "takes", len(subjects), "courses")
```

- Python agree skillfully responds with the following output:

```
John takes 2 courses
Vusi takes 3 courses
Jess takes 4 courses
Sarah takes 4 courses
Zuki takes 5 courses
```

- Now we'd like to ask how many students are taking ***CompSci***. This needs a counter, and for each student we need a second loop that tests each of the subjects in turn:

```
# Count how many students are taking CompSci
counter = 0
for name, subjects in students:
    for s in subjects:                    # A nested loop!
        if s == "CompSci":
            counter += 1
print("The number of students taking CompSci is", counter)
```

```
The number of students taking CompSci is 3
```

- A more concise of doing this would be the following:

```
counter = 0
for name, subjects in students:
    if "CompSci" in subjects:
        counter += 1
```

==========

Chapter 4: Functions (Pg 67– 69)

Functions with arguments and return values.

==========

Functions that require arguments

- Most functions require arguments: the arguments provide for generalization. For example, if we want to find the absolute value of a number, we have to indicate what the number is. Python has a built-in function for computing the absolute value:

Python Code	Output	
>>> abs(5)	5	In this example, the arguments to the abs function are 5 and-5.
>>> abs(-5)	5	

- Some functions take more than one argument. For example the built-in function pow takes two arguments, the base and the exponent.

Python Code	Output	
>>> pow(2, 3)	8	Inside the function, the values that are passed get assigned to variables called parameters.
>>> pow(7, 4)	2401	

- Another built-in function that takes more than one argument is max.

Python Code	Output
>>> max(7, 11)	11
>>> max(4, 1, 17, 2, 12)	17
>>> max(3 * 11, 5**3, 512- 9, 1024**0)	503

- ***max*** can be passed any number of arguments, separated by commas, and will return the largest value passed. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1.

Functions that return values

- All the functions in the previous section return values. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

Python Code	Output
biggest = max(3, 7, 2, 5) x = abs(3- 11) + 10 print ("Biggest = ",biggest, "\n x = ",x)	Biggest = 7 x = 18

- A ***function that returns a value*** is called a ***fruitful function***.
- The ***opposite of a fruitful function*** is ***void*** function — one that is not executed for its resulting value, but is executed because it does something useful.
- Even though void functions are not executed for their resulting value, Python always wants to return something. So if the programmer doesn't arrange to return a value, Python will automatically return the value None.
- How do we write our own fruitful function?
- In the exercises at the end of chapter 2 we saw the standard formula for compound interest, which we'll now write as a fruitful function:

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

```
def final_amount(p, r, n, t):
    a = p * (1 + r/n) ** (n*t)
    return a  # This is new, and makes the function fruitful.
# now that we have the function above, let us call it.
toInvest = float(input("How much do you want to invest?"))
fnl = final_amount(toInvest, 0.08, 12, 5)
print("At the end of the period you'll have", fnl)
```

- The return statement is followed an expression (a in this case). This expression will be evaluated and returned to the caller as the "fruit" of calling this function.
- We prompted the user for the principal amount. The type of to Invest is a string, but we need a number before we can work with it. Because it is money, and could have decimal places, we've used the float type converter function to parse the string and return a float.
- Notice how we entered the arguments for 8% interest, compounded 12 times per year, for 5 years.
- When we run this, we get the output ***At the end of the period you'll have 14898.457083.***
- The line ***toInvest = float(input("How much do you want to invest?"))*** also shows yet another ***example of composition*** — we can call a function like ***float***, and ***its arguments*** can be the ***results of other function calls*** (like input) that we've called along the way.

Notice something else very important here. The name of the variable we pass as an argument — toInvest — has nothing to do with the name of the parameter — p. It is as if p = toInvest is executed when final_amount is called. It doesn't matter what the value was named in the caller, in final_amount its name is p.

- These short variable names are getting quite tricky, so perhaps we'd prefer one of these versions instead:

```
def final_amount_v2(principal_amount, nominal_percentage_rate,
                                    num_times_per_year, years):
    a = principal_amount * (1 + nominal_percentage_rate /
                        num_times_per_year) ** (num_times_per_year*years)
    return a
def final_amount_v3(amount, rate, compounded, years):
    a = amount * (1 + rate/compounded) ** (componded*years)
    return a
def final_amount_v4(amount, rate, compounded, years):
    return amount * (1 + rate/compounded) ** (componded*years)
```

- They all do the same thing. Use your judgement to write code that can be best understood by other humans! Short variable names should generally be avoided, unless when short variables make more sense. This happens in particular with mathematical equations, where it's perfectly fine to use x, y, etc.

----------***END OF MODULE 1***----------