PYTHON PROGRAMMING

1BPLC105B/205B

Module-1

The way of the program: The Python programming language, what is a program? What is debugging? Syntax errors, Runtime errors, Semantic errors, Experimental debugging.

Variables, Expressions and Statements: 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: 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: Functions with arguments and return values.

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

1.1 The Python programming language

Python is a **high-level programming language**, just like C++, Java, C#, PHP, and Pascal. High-level languages are designed to be easy for humans to read, write, and understand. In contrast, **low-level languages** (such as machine language and assembly language) are difficult for humans but are the only languages computers can directly understand.

Because computers cannot run high-level language programs directly, such programs must first be **translated** into low-level language. In Python, this translation and execution are handled by a program called the **Python Interpreter**.

Most programs today are written in high-level languages because they offer many advantages. They take **less time to write**, are **shorter**, **easier to read**, and **less prone to errors**. Another major advantage is **portability**, meaning Python programs can run on different computers with little or no modification.

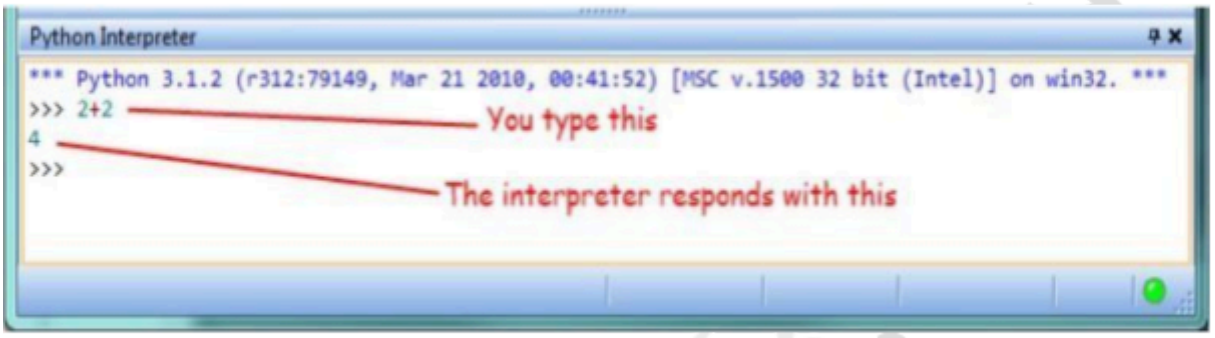
The Python Interpreter can be used in **two modes**:

1. **Immediate Mode (InteractiveMode):** You type commands directly into the interpreter, and it gives instant results. The >>> symbol is called the **Python prompt**, which shows that Python is ready to accept commands. This mode is useful for testing small pieces of code.
2. **Script Mode:** You write Python code in a file (called a **script**) and then run it using the interpreter. Scripts can be saved, reused, and are better for writing longer programs.

To write scripts, you need a **text editor**, such as Notepad, Notepad++, Sublime Text, Vim, or Emacs. These editors are different from word processors like MS Word because they work only with plain text.

Some software tools combine a text editor and the Python Interpreter. These are called **Development Environments** or **IDEs**. Popular Python IDEs include Spyder, Thonny, IDLE, and browser-based tools like Jupyter Notebook.

The choice of editor or IDE depends on personal preference or teacher recommendation. However, it is important to remember that **Python itself does not depend on the editor**. As long as the code is written with correct syntax, indentation, and spacing, Python will execute it correctly. The editor is only a tool to help the programmer.



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.

1.2 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, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

input Get data from the keyboard, a file, or some other device such as a sensor.

output Display data on the screen or send data to a file or other device such as a motor.

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.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated,is made up of instructions that look more or less like these. Thus, we can describe programming as the process of breaking a large, complex task into smaller and smaller subtasks until the

subtasks are simple enough to be performed with sequences of these basic instructions. That may be a little vague, but we will come back to this topic later when we talk about algorithms.

1.3 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.

1.4 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. So does this one For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of E. E. Cummings without problems. Python is not so forgiving. 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. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.5 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.

1.6 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.

1.7 Experimental debugging

Debugging is one of the most important skills a programmer learns. Although it can sometimes be frustrating, it is also one of the most interesting and intellectually challenging parts of programming. Debugging helps programmers understand how their programs actually work and why errors occur.

Debugging is similar to **detective work**. When a program does not work as expected, the programmer looks for clues in error messages, outputs, and program behavior. Using these clues, the programmer tries to find out what went wrong and how the error happened.

Debugging is also like an **experimental science**. The programmer forms a hypothesis about what might be causing the problem, then changes the program slightly and runs it again. If the result matches the prediction, the hypothesis was correct, and the program moves closer to working properly. If not, the programmer forms a new hypothesis and tries again.

The famous idea quoted by Sherlock Holmes applies well to debugging: when all impossible causes are eliminated, whatever remains—no matter how unlikely—must be the correct explanation. This approach encourages logical thinking and patience.

For many programmers, **programming and debugging are closely connected**. Programming is often seen as a process of writing a program and continuously fixing and improving it until it works as desired. Instead of writing a large program at once, it is better to start with a small working program and make small changes, debugging each step. This way, the program always remains functional.

A good example of this process is the development of **Linux**. Today, Linux is a powerful operating system with millions of lines of code, but it started as a very simple program written by Linus Torvalds to experiment with computer hardware. Over time, through continuous debugging and improvement, it evolved into the Linux operating system.

In conclusion, debugging is not just about fixing errors; it is a systematic way of thinking that helps programmers understand, improve, and successfully build software.

1.8 Values and data types

A **value** is one of the basic things that a program works with, such as a number or a piece of text. Examples of values include 4 (the result of 2 + 2) and "Hello, World!".

Each value in Python belongs to a specific **data type** (also called a **class**). A data type tells Python what kind of value it is and how it can be used.

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

Some common data types in Python are:

- **Integer (int)**: Whole numbers without decimal points, such as 4, 17, or 42000.

- **String (str)**: A sequence of characters (letters, numbers, symbols) enclosed in quotation marks, such as "Hello, World!".
- **Float (float)**: Numbers with decimal points, such as 3.2 or 5.75.

Python provides a built-in function called type () to find the data type of any value.

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

For example:

- type (17) returns int
- type(3.2) returns float
- type("Hello") returns str

Values like "17" or "3.2" may look like numbers, but because they are inside quotation marks, Python treats them as **strings**, not numbers.

```
>>> type("17")
<class 'str'>
>>> type("3.2")
<class 'str'>
```

In Python, strings can be written using:

- **Single quotes**: 'This is a string'

```
>>> 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 quotes**: "This is also a string"

```
>>> print('''"Oh no", she exclaimed, "Ben's bike is broken!"''')
"Oh no", she exclaimed, "Ben's bike is broken!"
>>>
```

- **Triple quotes** (''' ''' or """ """): Used for multi-line strings or strings containing both single and double quotes.

```
>>> message = """This message will
... span several
... lines."""
>>> print(message)
This message will
span several
lines.
>>>
```

Triple-quoted strings are especially useful when a string spans multiple lines or contains quotation marks inside it.

Python does not treat single-quoted, double-quoted, or triple-quoted strings differently internally. The quotation marks are **not part of the value**; they are only used to tell Python where the string begins and ends.

When Python displays a string, it usually shows it with **single quotes**, unless single quotes are already part of the string.

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

An important rule in Python is that **numbers should not contain commas or spaces**. For example:

- 42000 is a valid integer
- 42,000 is **not** treated as a number; Python interprets it as a pair of values (42, 0)

This shows that Python is a **strict formal language**, where even a small change in notation can completely change the meaning of the code.

1.9 Variables

One of the most powerful features of a programming language is the use of **variables**. A **variable** is a name that refers to a value stored in the computer's memory. Variables allow a program to store, remember, and manipulate data while the program is running.

In Python, values are given to variables using an **assignment statement**, which uses the assignment operator =.

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

Examples:

- message = "What's up, Doc?" assigns a string value to the variable message.
- n = 17 assigns an integer value to the variable n.
- pi = 3.14159 assigns a floating-point value to the variable pi.

The symbol = is called the **assignment operator** and should not be confused with the **equality operator**
==, which is used to compare two values. An assignment statement links the variable name on the left-hand side to the value on the right-hand side.

```
>>> 17 = n
File "<interactive input>", line 1
SyntaxError: can't assign to literal
```

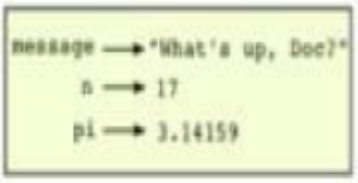
Tip: When reading or writing code, say to yourself "n is assigned 17" or "n gets the value 17". Don't say "n equals 17".

It is helpful to read assignment statements as:

- "n is assigned the value 17"
- Or "n gets the value 17" and **not** as "n equals 17".

An assignment must always have a variable on the left side. Writing 17 = n is invalid and results in a syntax error because a literal value cannot be assigned another value.

Variables can be visualized using a **state snapshot**, where the variable name points to its current value. This helps in understanding how values change during program execution.



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
```

When a variable name is typed in the interpreter, Python displays the value currently associated with that variable.

One important property of variables is that they are **changeable**. A variable can be assigned a new value at any time, and the new value replaces the old one.

```
>>> day = "Thursday"
>>> day
'Thursday'
```

(continues on next page)

(continued from previous page)

```
>>> day = "Friday"
>>> day
'Friday'
>>> day = 21
>>> day
21
```

Example:

- day = "Thursday"
- day = "Friday"
- day = 21

Here, the variable day is first assigned a string, then another string, and finally an integer. Python allows a variable to change its value and even its data type during execution.

Variables are widely used to help programs remember information, such as scores in a game, the number of missed calls on a phone, or user input. The ability to update variable values makes programs dynamic and useful.

1.10 Variable names and keywords

Rules for Variable Names in Python – Simplified Explanation

In Python, **variable names** can be of any length. They may contain **letters (a–z, A–Z)**, **digits (0–9)**, and the **underscore (_) symbol**. However, there are certain rules that must be followed.

A variable name **must begin with a letter or an underscore**, not with a digit. For example:

- myname valid
- _count valid
- 76trombones invalid (starts with a digit)

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
```

Although Python allows **uppercase letters** in variable names, by convention programmers usually use **lowercase letters**. Python is **case-sensitive**, which means Bruce and bruce are treated as **different variables**.

The underscore character _ is commonly used to separate words in variable names, especially when the name contains multiple words, such as:

- my_name
- price_of_tea_in_china

Some variable names that begin with an underscore have special meanings in Python, so beginners are advised to **start variable names with a letter**.

If an illegal character is used in a variable name, Python raises a **syntax error**. For example:

- more$ = 1000000 invalid (contains $)
- class = "Computer Science 101" invalid

The name class is invalid because it is a **Python keyword**.

Python Keywords

Keywords are reserved words in Python that define the language's syntax and structure. They **cannot be used as variable names**.

Some common Python key words include:

and, as, break, class, continue, def, elif, else, except, for, if, import, in, is, lambda, not, or, pass, raise, return, try, while, with, yield, True, False, None

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		

If Python shows an error for a variable name and the reason is unclear, it is a good idea to check whether the name is a **keyword**.

Choosing Meaningful Variable Names

Programmers usually choose **meaningful variable names** so that the code is easy for humans to read and understand. Good variable names act as **documentation** for the program.

However, beginners sometimes think that a variable name itself gives meaning to the computer. This is **not true**. The computer does **not** understand the meaning of words like average or pi. It only follows the instructions written by the programmer.

Caution: Beginners sometimes confuse "meaningful to the human readers" with "meaningful to the computer". So they'll wrongly think that because they've called some variable `average` or `pi`, it will somehow magically calculate an average, or magically know that the variable `pi` should have a value like 3.14159. No! The computer doesn't understand what you intend the variable to mean.

So you'll find some instructors who deliberately don't choose meaningful names when they teach beginners — not because we don't think it is a good habit, but because we're trying to reinforce the message that you — the programmer — must write the program code to calculate the average, and you must write an assignment statement to give the variable `pi` the value you want it to have.

```
e = 3.1415
ray = 10
size = e * ray ** 2
```

```
pi = 3.1415
radius = 10
area = pi * radius ** 2
```

The above two snippets do exactly the same thing, but the bottom one uses the right kind of variable names. For the computer there is no difference at all, but for a human, using the names and letters that are part of the conventional way of writing things make all the difference in the world. Using `e` instead of `pi` completely confuses people, while computers will just perform the calculation!

1.11 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 that we'll see shortly are while statements, for statements, if statements, and

import statements. (There are other kinds too!)

When you type a statement on the command line, Python executes it. Statements don't produce any result.

1.12 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. We've previously seen the print and the type functions, so this is our third example of a function!

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
```

(continues on next page)

```
5
>>> y
3.14
```

1.13 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
```

Take care that you choose the correct flavor of the division operator. If you're working with expressions where you

need floating point values, use the division operator that does the division accurately.

1.14 Type converter functions

Here we'll look at three more 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 floating 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")
```

This last case doesn't look like a number — what do we expect?

```
>>> float(17)
17.0
>>> float("123.45")
123.45
```

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:

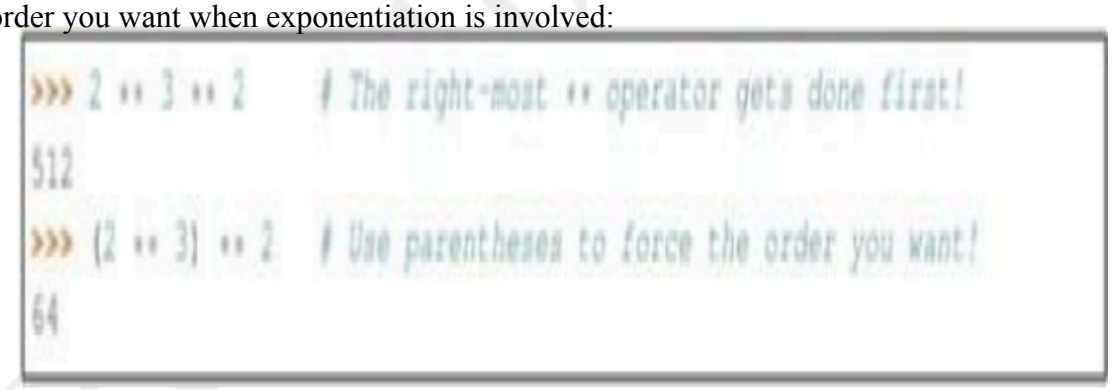
```
>>> str(17)
'17'
>>> str(123.45)
'123.45'
```

1.15 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 PEM DAS is a useful way to remember the order of operations:

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. 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. Exponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.
3. Multiplication and both Division operators have the same precedence, which is higher than Addition and Subtraction, 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. If the operations had been evaluated from right to left, the result would have been 6-(3+2), which is 1. (The acronym PEDMAS could mislead you to thinking that division has higher precedence than multiplication, and addition is done ahead of subtraction - don't be misled. Subtraction and addition are at the same precedence, and the left-to-right rule applies.)
   Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator **, so a useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:



   The immediate mode command prompt of Python is great for exploring and experimenting with expressions like this.

1.16 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:

```
1 fruit = "banana"
2 baked_good = " nut bread"
3 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

1.17 Input

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

```
1 name = 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 we saw earlier.

1.18 Composition

So far, we have looked at the elements of a program — variables, expressions, statements, and function calls in isolation, without talking about how to combine them. 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

$$\text{Area} = \pi R^2$$

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.

1.19 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)
```

1.20 Iteration

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a set of statements is called iteration. Because iteration is so common, Python provides several language features to make it easier.

1.20.1 Assignment

As we have mentioned previously, 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. (In some programming languages, a different symbol is used for assignment, such as <- or :=, to avoid confusion. Some people also think that variable was an unfortunae word to choose, and instead we should have called them assignables. Python chooses to follow common terminology and token usage, also found in languages like C, C++, Java, and C#, so we use the tokens = for assignment, == for equality, and we talk of variables.

1.20.2 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. It is 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
```

1.20.3 The for loop revisited

The for loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. We saw this example before:

```
for friend in ["Joe", "Zoe", "Zuki", "Thandi", "Paris"]:
    invite = "Hi " + friend + ".  Please come to my party!"
    print(invite)
```

Running through all the items in a list is called traversing the list, or traversal. 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)
```

1.20.4 The while statement

The while statement in Python is used to **repeat a block of code as long as a condition remains true**. It is especially useful when the number of repetitions is **not known in advance**.

```
while <CONDITION>:
    <STATEMENT>
```

```
n = 6

current_sum  = 0
i = 0
while i <= n:
    current_sum += i
    i += 1
print(current_sum)
```

This program calculates the **sum of numbers from 0 to 6**.

You can read the while loop almost like English:

- *While i is less than or equal to n, keep executing the loop body.*
- Each time, add i to current_sum.
- Then increase i by 1.
- When i becomes greater than n, the loop stops and the final sum is printed.

How a while Loop Executes (Flow of Execution)

1. The condition (i <= n) is evaluated.
2. If the condition is **False**, the loop ends and the program moves to the next statement.
3. If the condition is **True**, the statements inside the loop body are executed.
4. After executing the body, the program goes back and checks the condition again.
5. This process repeats until the condition becomes False.

The **loop body** consists of all statements that are **indented** under the while keyword.

If the condition is False the very first time it is checked, the loop body is **never executed**.

Loop Termination and Infinite Loops

The body of a while loop must **change one or more variables** involved in the condition. If the condition never becomes False, the loop will run forever. This situation is called an **infinite loop**.

In this example:

- n is a fixed value.
- i increases by 1 each time.
- Eventually, i becomes greater than n, so the loop terminates.

In more complex programs, it is sometimes difficult to know whether a while loop will ever stop.

Comparison: while Loop vs for Loop

The same task can be written more simply using a for loop:

```
n = 6

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

Here, the for loop automatically:

- Initializes the loop variable
- Checks the stopping condition
- Updates the loop variable

This makes for loops **simpler and less error-prone** when iterating over a known range.

Why Use a while Loop Then?

Although for loops are easier in many cases, while loops provide **more control**. They are useful when:

- The number of iterations is not known beforehand
- Loop termination depends on a condition that changes dynamically
- More flexible logic is required

- while loop runs as long as the condition is True
- Condition is checked **before** each iteration
- Loop body must change the condition variable
- Risk of **infinite loop** if condition never becomes False

- for loops are simpler when the range is known

1.20.5 The Collatz 3n + 1 sequence

The **Collatz sequence** is a famous mathematical sequence that has puzzled mathematicians for many years. Even today, no one has been able to fully prove why it behaves the way it does.

Rule of the Collatz Sequence

Start with any **positive integer n** and repeatedly apply the following rules:

- If n is **even**, divide it by 2 → n = n // 2
- If n is **odd**, multiply it by 3 and add 1 → n = 3 * n +

1 The sequence **continues until n becomes 1**.

```
n = 1027371

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

Explanation of the Code

- The loop runs **while n is not equal to 1**
- n % 2 == 0 checks whether n is even
- Depending on whether n is even or odd, the appropriate rule is applied
- The print(..., end=", ") keeps printing numbers on the **same line**
- When n finally becomes 1, the loop stops and prints the final value

Why Is the Collatz Sequence Interesting?

During the sequence:

- Sometimes n **increases**
- Sometimes n **decreases**

Because of this unpredictable behavior, it is **not obvious** whether the sequence will always reach 1. For so me numbers, termination is easy to prove. For example, if n starts as a **power of 2** (like 16), it will always be even and quickly reduce to 1. However, for many other numbers, the sequence can take **a very long time** before reaching 1. Some small starting numbers require **more than 100 steps**.

The Collatz Conjecture

The **Collatz conjecture** (also called the **3n + 1 conjecture**) states:

All positive integers will eventually reach 1 if the Collatz rules are applied repeatedly.

So far:

- Computers have tested **very large numbers**
- Every tested number **eventually reaches 1**
- But **no mathematical proof or disproof**

exists Because of this, the conjecture remains

unsolved. **Cycles in the Sequence**

If the process does **not stop at 1**, the sequence enters a repeating cycle:

1 → 4 → 2 → 1 → 4 → 2 → ...

One possibility is that **other cycles might exist**, but none have been found yet.

Choosing Between for and while Loops

Use a for loop when:

- You know **in advance** how many times the loop will run Examples:
- Printing tables
- Iterating through a list
- Finding prime numbers up to a fixed

limit This is called **definite iteration**.

Use a while loop when:

- You do **not know in advance** how many iterations are needed Examples:
- Running until a condition becomes true
- Problems like the Collatz

sequence This is called **indefinite**

iteration.

1.20.6 Tracing a program

To write effective programs and clearly understand how a program works, a programmer must learn how to **trace a program**. Tracing means **manually following the execution of a program step by step**, just like a computer would, and keeping track of:

- The **values of all variables**
- The **output produced** after each statement

Tracing helps build a strong mental model of program execution and is extremely useful for **debugging**.

How Tracing Works

When tracing a program, we:

1. Start with the initial values of variables
2. Follow the flow of execution line by line
3. Update variable values after each instruction
4. Record any output generated

A common technique is to draw a **table** with:

- One column for each variable
- One column for output

Tracing the Collatz Program (n = 3)

Let us trace the Collatz sequence program when n = 3.

Initial State

- n = 3
- Since n != 1, the while loop starts

Step-by-Step Execution

- 3 is printed
- 3 % 2 == 0 → False, so the **else** branch runs
- New value of n = 3 * 3 + 1 = 10

Then the loop repeats:

n	Output Printed So Far
3	3,
10	3, 10,
5	3, 10, 5,
16	3, 10, 5, 16,
8	3, 10, 5, 16, 8,
4	3, 10, 5, 16, 8, 4,
2	3, 10, 5, 16, 8, 4, 2,
1	3, 10, 5, 16, 8, 4, 2, 1.

When n becomes 1, the condition n != 1 becomes **False**, and the loop terminates. The final value 1 is printed outside the loop.

What We Learn from Tracing

Tracing, though slow and sometimes tedious, helps us:

- Understand **how and why variable values change**
- Predict how many times a loop will run
- Detect logical errors
- Understand output

behavior From this trace, we

observe:

- Once n becomes a **power of 2**, the sequence decreases quickly
- The final value 1 is **not printed inside the loop**, which is why a separate print statement is needed after the loop
- The program's behavior becomes more predictable at certain stages

Why Tracing Is Important

Although computers execute programs automatically, **humans must understand the logic** behind the execution. Tracing helps programmers:

- Debug programs
- Improve efficiency
- Gain confidence in program correctness

Tracing is an **essential skill** for every programmer, especially begissnners.

1.20.7 Counting digits

This section explains how a program can **count the number of digits** in a positive integer using a while loop.

Counting Total Digits in a Number

```
n = 3029
count = 0
while n != 0:
    count = count + 1
    n = n // 10
print(count)
```

How the Program Works

- The variable count is initialized to 0
- Each time the loop runs:
    - count is increased by 1

- The last digit of n is removed using integer division (n // 10)

- The loop continues until n becomes 0
- The final value of count represents the **number of digits** in the original number

This technique is called a **counter pattern**, where a variable is incremented each time a loop executes.

n	count
3029	0
302	1
30	
3	3
2	
0	4

```
n = 2574301453
count = 0
while n > 0:
    digit = n % 10
    if digit == 0 or digit == 5:
        count = count + 1
    n = n // 10
print(count)
```

Explanation

- n % 10 extracts the **last digit** of the number
- The if condition checks whether the digit is 0 or 5
- If true, the counter is incremented
- The number is shortened by removing the last digit
- The loop continues until all digits are processed

This program counts **how many times the digits 0 or 5 appear** in the number.

Important Question: What if n = 0?

If n = 0, the loop condition while n > 0 is **False immediately**, so:

- The loop body never executes
- count remains 0
- The program prints 0, not 1

Why Does This Happen?

This happens because:

- The code assumes n is a **positive integer**
- The number 0 has **one digit**, but the loop never runs
- **Counter pattern**
- Integer division (//)
- Modulus operator (%)
- Loop control using conditions
- Importance of handling edge cases

1.20.8 Help and meta-notation

Python provides **extensive built-in documentation** for its language features, functions, and libraries. Programmers can use this documentation to understand how functions work, what arguments they take, and how they should be used. This help is available through official Python documentation websites and also through built-in help tools.

When reading Python documentation, you will often see **special symbols and formatting** that are not part of actual Python code. These are called **meta-notation**. Meta-notation is used to **describe Python syntax**, not to be typed directly into programs.

Square Brackets [ ] in Documentation

Square brackets indicate that something is **optional**.

Example from documentation:

range([start], stop[, step])

This means:

- stop is **mandatory**
- start is **optional**
- step is **optional**

So, range() can be used with:

- 1 argument: range(stop)
- 2 arguments: range(start, stop)
- 3 arguments: range(start, stop, step)

The documentation also tells us that all arguments to range() must be **integers** and that the sequence can increase or decrease depending on the step.

Bold and Italics in Meta-Notation

Documentation often uses **bold text** and *italic text*:

- **Bold text** represents **exact Python keywords or symbols** that must be typed exactly as shown.
- *Italic text* represents a **placeholder**, meaning you should replace it with something valid of that type.

For example:

for variable in list:

Here:

- for, in, and : are typed exactly as shown
- variable can be any valid variable name
- list can be any valid iterable

Ellipses (...) in Documentation

Ellipses mean **"zero or more**

items". Example:

print([object, ...])

This tells us:

- The print function can take **any number of arguments**
- Arguments are separated by commas
- It is even valid to call print() with no arguments

Why Meta-Notation Is Important

Meta-notation allows documentation to:

- Explain complex syntax **clearly and concisely**
- Show patterns instead of listing every possible case
- Help programmers correctly use functions and language features

Understanding meta-notation helps programmers **read documentation effectively**, which is a critical skill in programming.

1.20.9 Tables

One of the important uses of **loops** in programming is to **generate tables of values**. Before computers were common, people manually calculated values such as logarithms, sines, and cosines and wrote them in printed tables. This process was slow, boring, and often contained errors.

With the arrival of computers, it became easy to generate such tables automatically and accurately. Eventually, calculators and computers became so common that printed tables were no longer needed.

However, tables are still used internally by computers for **approximate calculations**, such as in floating-point arithmetic. In fact, one of the most famous computer bugs occurred due to an error in the floating-point division table of the Intel Pentium processor.

Even though tables are not as important today, they are still an excellent example to demonstrate **iteration using loops**.

```
for x in range(13):   # Generate numbers 0 to 12
    print(x, "\t", 2**x)
```

This program prints:

- The value of x in the first column
- The value of 2 raised to the power x in the second column

Escape Sequences

The string "\t" represents a **tab character**. The backslash (\) begins an **escape sequence**, which represents characters that are not visible on the screen.

Common escape sequences:

- \t → Tab
- \n → New line

To represent a **backslash** itself in a string, we use:

"\\"

How Tab Characters Work

When text is printed on the screen, an invisible pointer called the **cursor** keeps track of where the next character will appear. Normally, after a print statement, the cursor moves to the beginning of the next line. The tab character (\t) moves the cursor to the **next tab stop**. This helps align text neatly into columns, regardless of how many digits appear in each column.

```
0       1
1       2
2       4
3       8
4       16
5       32
6       64
7       128
8       256
9       512
10      1024
11      2048
12      4096
```

Because of the tab character, the second column remains properly aligned even when the first column has numbers with different digit lengths.

1.20.10 Two-Dimensional Tables

A **two-dimensional table** is a table in which values are arranged in rows and columns, and each value is found at the **intersection of a row and a column**. A **multiplication table** is a common and easy example of a two-dimensional table.

For example, a multiplication table shows how each number (row) is multiplied by another number (column).

Printing One Row of a Multiplication Table

Before printing a full table, it is helpful to start with a **single row**. The following program prints the multiples of 2 from 1 to 6 on a single line.

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

Explanation of the Program

- range(1, 7) generates numbers from 1 to 6
- The variable i takes each value from the range
- 2 * i calculates the multiple of 2
- end=" " prevents the print function from moving to a new line and instead prints a space after each value
- The final print() moves the cursor to the next line after the loop finishes

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

All values appear on the **same line**, separated by spaces.

Why This Is Important

This example demonstrates:

- Use of a **for loop**
- Control of output formatting using end
- A building block for creating **full two-dimensional tables**

To create a complete multiplication table (rows and columns), we would later use **nested loops**, where:

- One loop controls the rows
- Another loop controls the columns

This process of improving and generalizing code step by step is an important programming practice.

1.20.11 The 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:

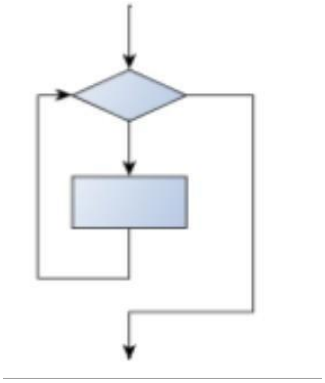
```
for i in [12, 16, 17, 24, 29]:
    if i % 2 == 1:  # If the number is odd
        break       #  ... immediately exit the loop
    print(i)
print("done")
```

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.



1.20.12 Other Flavours of Loops – Simplified Explanation

In programming, loops can differ based on **where the exit condition is tested**. Some languages provide separate loop constructs for each case, but **Python uses only the while loop**, combined with if and break, to handle all these variations.

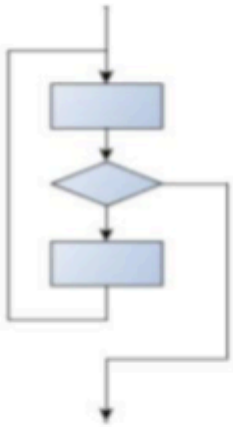
The three common loop "flavours" are:

1. **Pre-test loop** – condition tested before the loop body
2. **Middle-test loop** – condition tested in the middle of the loop
3. **Post-test loop** – condition tested at the end of the loop

1. Middle-Test Loop

A **middle-test loop** checks the exit condition **after doing some work**, but **before finishing the loop body**. This pattern is very common in **interactive programs**, where input must be read first before deciding whether to continue.

The middle-test loop flowchart



```
total = 0
while True:
    response = input("Enter the next number. (Leave blank to end)")
    if response == "" or response == "-1":
        break
    total += int(response)
print("The total of the numbers you entered is ", total)
```

How It Works

- while True: creates an infinite loop
- Input is taken first
- If the user enters a blank line or -1, the loop exits using break
- Otherwise, the number is added to total

Here, the **exit decision happens in the middle** of the loop body.

Why while True: Is Used

The condition True is always true, so the loop would normally run forever. This is a **Python idiom**—a commonly accepted programming pattern. Since the loop cannot end naturally, the programmer must explicitly exit using break. Modern compilers and interpreters recognize this as a **dummy condition**, so they optimize it efficiently.

2. Post-Test Loop

A **post-test loop** checks its exit condition **after the loop body has executed at least once**. This ensures that the loop body **always runs at least one time**.

Python does not have a built-in post-test loop, but we can simulate it using while True: and break.

Example: Playing a Game at Least Once

```
while True:
    play_the_game_once()
    response = input("Play again? (yes or
    no)") if response != "yes":
        break

print("Goodbye!")
```

Explanation

- The game is played **before** asking whether to continue
- The exit condition is tested **at the end**
- This guarantees the game runs at least once

3. Choosing Where the Exit Test Should Be

When designing a loop, ask:

- **When should the loop stop?**
- Should the test be:

- **Before** the first iteration? → Pre-test loop
- **In the middle** of each iteration? → Middle-test loop
- **After** at least one iteration? → Post-test loop

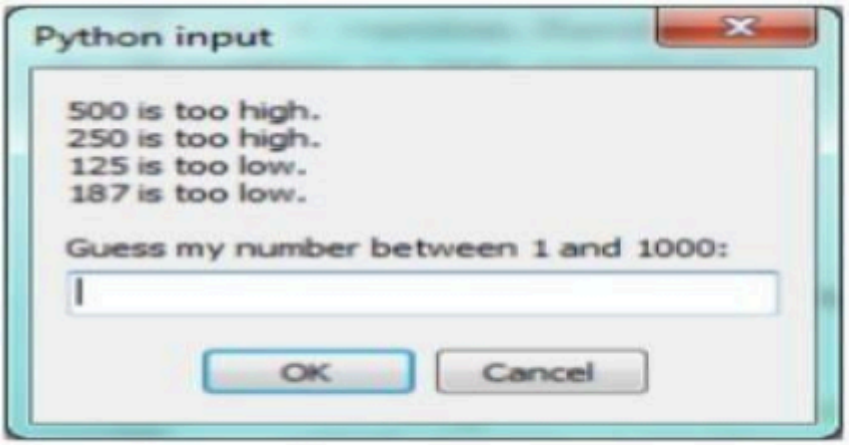
Interactive programs and file-processing programs often require **middle-test or post-test loops**, because the decision to stop can only be made **after reading input**.

1.12.13 An example

The following program implements a simple guessing game:

```
1  import random                    # We cover random numbers in the
2  rng = random.Random()            #  modules chapter, so peek ahead if you
   ↪want. "rng" stands for "random number generator".
3  number = rng.randrange(1, 1000) # Get random number between [1 and 1000).
4
5  guesses = 0
6  message = ""
7
8  while True:
9      guess = int(input(message + "\nGuess my number between 1 and 1000: "))
10     guesses += 1
11     if guess > number:
12         message += str(guess) + " is too high.\n"
13     elif guess < number:
14         message += str(guess) + " is too low.\n"
15     else:
16         break
17
18 input("\n\nGreat, you got it in "+str(guesses)+" guesses!\n\n")
```

This program makes use of the mathematical law of trichotomy (given real numbers a and b, exactly one of these three must be true: a > b, a < b, or a == b). At line 18 there is a call to the input function, but we don't do anything with the result, not even assign it to a variable. This is legal in Python. Here it has the effect of popping up the input dialog window and waiting for the user to respond before the program terminates. Programmers often use the trick of doing some extra input at the end of a script, just to keep the window open.Also notice the use of the message variable, initially an empty string, on lines 6, 12 and
14. Each time through the loop we extend the message being displayed: this allows us to display the program's feedback right at the same place as we're asking for the next guess.



1.20.14 The 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")
```

This prints:

```
12
16
24
30
done
```

1.20.15 Paired Data

In Python, data can be grouped together to form **pairs**, which allow related pieces of information to be stored and processed together. A pair is created by placing two values inside **parentheses**, separated by a comma. Such a structure is commonly called a **tuple**.

```
year_born = ("Paris Hilton", 1981)
```

We can put many pairs into a list of pairs:

```
celebs = [("Brad Pitt", 1963), ("Jack Nicholson", 1937),
                               ("Justin Bieber", 1994)]
```

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

```
print(celebs)
print(len(celebs))
```

```
[("Brad Pitt", 1963), ("Jack Nicholson", 1937), ("Justin Bieber", 1994)]
3
```

Here:

- "Paris Hilton" is a name
- 1981 is the year of birth Both values together form one paired data item

How This Loop Works

- The loop runs once for each pair in the list
- On each iteration:
  - name gets the first value from the pair
  - year gets the second value from the pair
- The condition checks the year and prints the name if it is less

This is different from earlier loops that used only **one loop** than 1980

variable. **Why Paired Data Is Useful**

Paired data helps in:

- Organizing related information
- Writing cleaner and more readable code
- Processing structured data efficiently

This approach is commonly used in:

- Student records (name, marks)
- Product lists (item, price)

- Databases and real-world data processing

1.20.16 Nested Loops for Nested Data

In Python, data is often **nested**, meaning that one data structure exists inside another. To process such data, we use **nested loops**—that is, a loop inside another loop.

In this example, we have a list of students. Each student's data consists of:

- A **name**
- A **list of subjects** the student is enrolled in

```
for name, year in celebs:
    if year < 1980:
        print(name)
```

```
Brad Pitt
Jack Nicholson
```

Here:

- students is a **list**
- Each element is a **pair (tuple)**
- The second item in each pair is **another list**

This is an example of **nested data**.

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:

```
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")
```

Explanation

- The loop runs once for each student
- name stores the student's name
- subjects stores the list of subjects
- len(subjects) counts how many subjects the student is taking

Counting Students Taking CompSci (Using Nested Loops)

To find how many students are taking **CompSci**, we must check each student's list of subjects. This requires a **nested loop**.

```
# 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
```

Explanation

- The outer loop goes through each student
- The inner loop goes through each subject of that student
- If "CompSci" is found, the counter is increased

Output

The number of students taking CompSci is 3

A More Concise Approach

Python provides a simpler way to do this using the in operator:

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

This works because:

- "CompSci" in subjects checks the entire list at once
- It avoids the need for an inner loop
- The code is shorter and easier to read

Why Nested Loops Are Important

Nested loops are used when:

- Data is nested (lists inside lists, pairs containing lists)

- Each item in the outer structure contains multiple inner items
- Detailed inspection of inner data is

required They are common in:

- Student databases
- Product catalogs
- Movie–actor lists
- Music playlists

1.20.17 Newton's method for finding square roots

Loops are often used in numerical computations where we **start with an approximate answer and improve it step by step**. One famous method for doing this is **Newton's method**, which is used to calculate square roots.

Before calculators and computers existed, people computed square roots manually. Newton's method is especially powerful because it **converges very quickly**, meaning it reaches an accurate result in only a few steps.

Basic Idea of Newton's Method

To find the square root of a number n, we:

1. Start with an **initial guess** (approximation)
2. Improve the guess using the formula:

```
better = (approximation + n/approximation)/2
```

Each time this formula is applied, the result gets **closer to the actual square root**. **Why Iteration Is Needed**

We do not know in advance:

- How many times the formula must be applied
- When the approximation is "good enough"

This makes it an **indefinite iteration problem**, which is best handled using a **while loop**.

Stopping Condition ("Close Enough")

Exact equality between two real numbers is unreliable in computers because real numbers are stored approximately. Instead, we stop when the difference between two successive guesses is **very small**.

This is done using a **threshold** value:

```
threshold = 0.001
if abs(a-b) < threshold:  # Make this smaller for better accuracy
    break
```

Python Program Using Newton's Method

```
n = 8
threshold = 0.001
approximation = n/2     # Start with some or other guess at the answer
while True:
    better = (approximation + n/approximation)/2
    if abs(approximation - better) < threshold:
        print(better)
```

(continues on next page)

Explanation of the Code

- Start with an initial approximation (n / 2)
- Improve it using Newton's formula
- Compare the old and new approximations
- Stop when the difference is smaller than the threshold
- Print the final approximation

This is a **middle-exit loop**,

because:

- Some work is done first
- The exit condition is checked in the middle
- The loop exits using break

Why Newton's Method Is Efficient

- Converges very fast
- Needs only a few iterations
- Used internally by calculators and computers
- Works well for large and small numbers

Example: sqrt(25)

If you start with an initial guess (say 12.5) and repeatedly apply Newton's formula, you will see that:

- The value quickly approaches 5
- Only a few iterations are needed to reach high accuracy

1.20.18 Algorithms

Newton's method is an example of an **algorithm**. An algorithm is a **step-by-step, mechanical process** used to solve a **class of problems**, not just one specific problem. In this case, Newton's method provides a general way to compute **square roots of any number**.

What Is Algorithmic Knowledge?

Some types of knowledge are **algorithmic**, meaning they follow a fixed procedure. Examples include:

- Addition with carrying
- Subtraction with borrowing
- Long division
- Solving Sudoku puzzles using fixed steps

These processes follow clear rules and can be repeated for many similar problems.

What Is Not Algorithmic Knowledge?

Other kinds of knowledge rely on **memorization** rather than procedures. Examples include:

- Remembering historical dates
- Memorizing multiplication tables

These do not involve a step-by-step process for generating answers.

Key Characteristics of Algorithms

- Algorithms are **mechanical**: they do not require intelligence or creativity to execute
- Each step follows logically from the previous one
- They are designed to solve a **general category of problems**
- They can be executed by **machines or computers**

Once an algorithm is designed, a computer can carry it out efficiently and accurately.

Importance of Algorithmic Thinking

The idea that complex problems can be solved through **simple, step-by-step procedures** is one of the greatest breakthroughs in human history. With computers executing algorithms, humans can solve problems at a scale and speed that was previously impossible.

Algorithmic or **computational thinking**—using automation and algorithms to approach problems—is transforming modern society. Some experts believe this shift will have an impact even greater than the invention of the **printing press**.

Designing Algorithms

While executing an algorithm may be repetitive or boring, **designing algorithms** is:

- Intellectually challenging

- Creative
- A central part of

programming Programmers must

decide:

- What steps are needed
- In what order they should occur
- When the process should stop

Limits of Algorithms

Interestingly, some tasks that humans perform easily are **very difficult to express algorithmically**. A good example is **understanding natural language**. Humans understand language naturally, but creating a complete step-by-step algorithm to explain how this happens is extremely challenging and still an open problem in computer science.

1.21 Functions that require arguments

Most functions in Python **require arguments**. Arguments are the values that we pass to a function so that it can perform its task. Arguments allow functions to be **generalized**, meaning the same function can work with many different inputs.

```
>>> abs(5)
5
>>> abs(-5)
5
```

Here, 5 and -5 are the **arguments** passed to the function. The function uses these values to compute the result.

Functions with More Than One Argument

Some functions need **more than one argument**.

Example: pow() Function

The pow() function takes **two arguments**:

- The **base**
- The **exponent**

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

and the exponent. Inside the function, the values that are passed get assigned to variables called parameters.

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

- Arguments provide **input** to functions
- Functions use arguments to produce results
- Parameters are the variable names used **inside** the function
- Some functions take one argument (abs)
- Some take two (pow)
- Some take many (max)
- Arguments can be values or expressions

1.22 Functions that return values

Functions in Python can be divided into two main types based on whether they **return a value** or not.

Functions That Return Values (Fruitful Functions)

A function that returns a value is called a **fruitful function**. When such a function is called, it produces a result that can:

- Be stored in a variable, or
- Be used as part of an expression

```
1 biggest = max(3, 7, 2, 5)
2 x = abs(3 - 11) + 10
```

Here:

- max() returns the largest value
- abs() returns the absolute value
- The returned values are used in assignments and expressions

Void Functions

Some functions are written **not to compute a value**, but to **perform an action**, such as drawing or printing.
For example, a function like draw_square() is executed to make the turtle draw a shape, not to return a number.

Such functions are called **void functions**.

However, in Python:

- Every function **always returns something**
- If no return statement is used, Python automatically returns None

Writing Our Own Fruitful Function

To make a function return a value, we use the **return statement**.

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

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):
    """
      Apply the compound interest formula to p
       to produce the final amount.
    """

    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?"))
```

(continues on next page)

- 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.