

CST 362	PROGRAMMING IN PYTHON	Category	L	T	P	Credit	Year of Introduction
		PEC	2	1	0	3	2019

Prerequisite: Basic knowledge in Computational Problem Solving, A course in any programming language.

Mark Distribution

Total Marks	CIE Marks	ESE Marks	ESE Duration
150	50	100	3

Continuous Internal Evaluation Pattern:

Attendance : 10 marks

Continuous Assessment Test : 25 marks

Continuous Assessment Assignment : 15 marks

Course Outcomes

CO1	Write, test and debug Python programs (Cognitive Knowledge level: Apply)
CO2	Illustrate uses of conditional (if, if-else and if-elif-else) and iterative (while and for) statements in Python programs. (Cognitive Knowledge level: Apply)
CO3	Develop programs by utilizing the Python programming constructs such as Lists, Tuples, Sets and Dictionaries. (Cognitive Knowledge level: Apply)
CO4	Develop graphical user interface for solutions using Python libraries. (Cognitive Knowledge level: Apply)
CO5	Implement Object Oriented programs with exception handling. (Cognitive Knowledge level: Apply)
CO6	Write programs in Python to process data stored in files by utilizing Numpy, Matplotlib, and Pandas. (Cognitive Knowledge level: Apply)

Text Books:

1. Kenneth A Lambert., Fundamentals of Python : First Programs, 2/e, Cengage Publishing, 2016
2. Wes McKinney, Python for Data Analysis, 2/e, Shroff / O'Reilly Publishers, 2017
3. Flask: Building Python web services, Jack Stouffer, Shalabh Aggarwal, Gareth Dwyer, PACKT Publishing Limited, 2018

Module -1 (Programming Environment and Python Basics)

Getting started with Python programming –

Interactive shell

IDLE

iPython Notebooks

Detecting and correcting syntax errors

How Python works.

The software development process

A case study.

Basic coding skills –

Strings

Assignment and comments

Numeric data types and character sets

Expressions

Using inbuilt functions and modules.

Control statements

Iteration with for/while loop

Formatting text for output

A case study

Selection structure

Conditional iteration with while

A case study

Testing control statements

Lazy evaluation.

Getting Started with Python Programming

Guido van Rossum Dutch programmer invented the Python programming language in the early 1990s.

Python is a high-level, general-purpose programming language for solving problems on modern computer systems.

The language and many supporting tools are free, and Python programs can run on any operating system.

Download Python, its documentation, and related materials from www.python.org.

Running Code in the Interactive Shell

Python is an interpreted language, and can run simple Python expressions and statements in an interactive programming environment called the shell.

The easiest way to open a Python shell is to launch the **IDLE (Integrated DeveLopment Environment)**.

This is an integrated program development environment that comes with the Python installation.

A shell window contains an opening message followed by the special symbol

```
>>>
```

called a shell prompt.

The cursor at the shell prompt waits for user to enter a Python command.

<pre>>>> 3 + 4 7 >>> 3 3 >>> "Python is really cool!" 'Python is really cool!' >>> name = "Ken Lambert" >>> name 'Ken Lambert' >>> "Hi there, " + name 'Hi there, Ken Lambert' >>> print('Hi there') Hi there >>> print("Hi there,", name) Hi there, Ken Lambert</pre>	<pre># Simple arithmetic # The value of 3 is # Use a string for text # Give a variable a value # The value of name is # Create some new text # Output some text # Output two values</pre>
--	---

Color	Type of Element	Examples
Black	Inputs in the IDLE shell Numbers Operator symbols Variable, function, and method references Punctuation marks	<code>67, +, name, y = factorial(x)</code>
Blue	Outputs in the IDLE shell Function, class, and method names in definitions	<code>'Ken Lambert', def factorial(n)</code>
Green	Strings	<code>"Ken Lambert"</code>
Orange	Keywords	<code>def, if, while</code>
Purple	Built-in function names	<code>abs, round, int</code>
Red	Program comments Error messages in the IDLE shell	<code># Output the results ZeroDivisionError: division by zero</code>

Table 1-1

Color-coding of Python program elements in IDLE

Input, Processing, and Output

Programs accept inputs from some source, process these inputs, and then finally output results to some destination

In terminal-based interactive programs, the input source is the keyboard, and the output destination is the terminal display.

Output of a value by using the print function

`print(<expression>)`

```
>>> print ("Hi there")  
Hi there
```

```
>>> print(3+4)  
7
```

Print function that includes two or more expressions separated by commas.

In such a case, the print function evaluates the expressions and displays their results, separated by single spaces, on one line.

```
print(<expression>, ..., <expression>)
```

```
>>> print("sum of ",3," and ",4," is ", 3+4)
sum of 3 and 4 is 7
```

Print function always ends its output with a **newline**

To begin the next output on the same line as the previous one

```
print(<expression>, end = "")
```

input function

Stop and wait for the user to enter a value from the keyboard.

When the user presses the return or enter key, the function accepts the input value and makes it available to the program.

A program that receives an input value in this manner typically saves it for further processing.

```
>>> name = input("Enter your name: ")
Enter your name: Ken Lambert
>>> name
'Ken Lambert'
>>> print(name)
Ken Lambert
```

The input function always builds a string from the user's keystrokes and returns it to the program.

After inputting strings that represent numbers, the programmer must convert them from strings to the appropriate numeric types.

In Python, there are two **type conversion functions**

- **int (for integers)**
- **float (for floating point numbers).**

```
>>> first = int(input("Enter the first number: "))
Enter the first number: 23
>>> second = int(input("Enter the second number: "))
Enter the second number: 44
>>> print("The sum is", first + second)
The sum is 67
```

To compose and execute programs as script file:

1. Select the option New Window from the File menu of the shell window.
2. In the new window, enter Python expressions or statements on separate lines, in the order in which you want Python to execute them.
3. At any point, you may save the file by selecting File/Save. If you do this, you should use a **.py** extension.
4. To run this file of code as a Python script, select Run Module from the Run menu or press the F5 key.

area.py - F:/ACADEMICS/Books and Notes/EVEN/S6/Python/Pro

File Edit Format Run Options Window Help

```
length=int(input("Enter the length"))  
width=int(input("Enter the width"))  
area=length*width  
print("area is ",area)
```

...

=== RESTART: F:/ACADEMICS/Books and Notes/EVEN/S6/Python/Programs/area.py ===

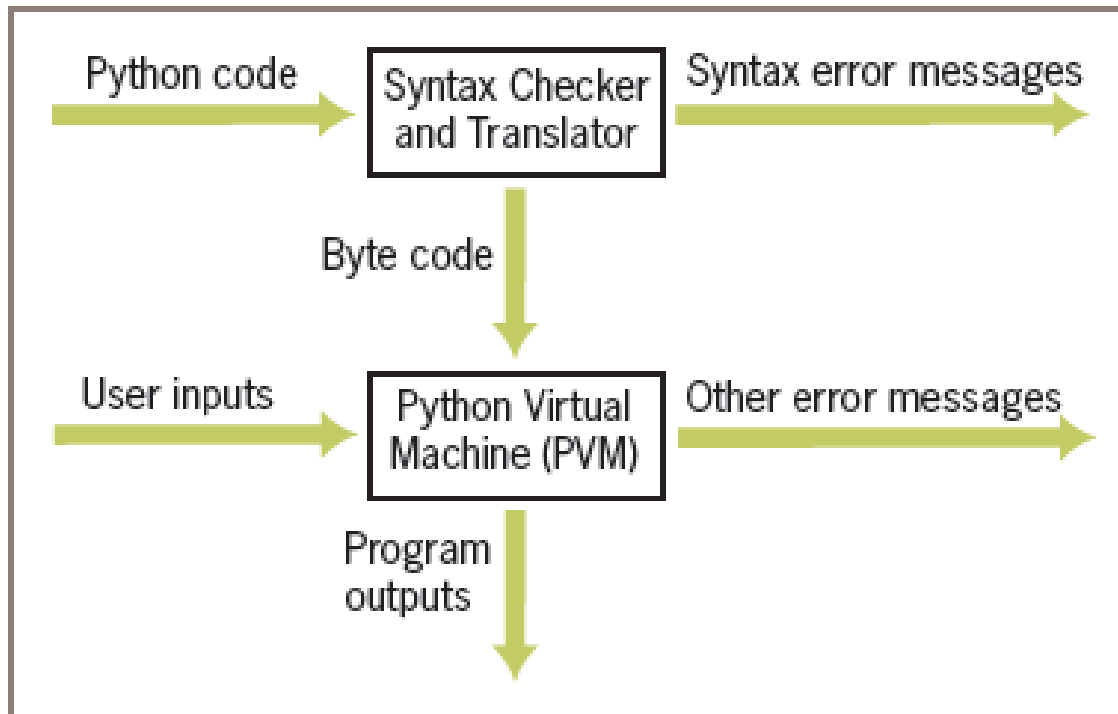
Enter the length5

Enter the width4

area is 20

>>>

How Python Works



Steps in interpreting a Python program

Detecting and Correcting Syntax Errors

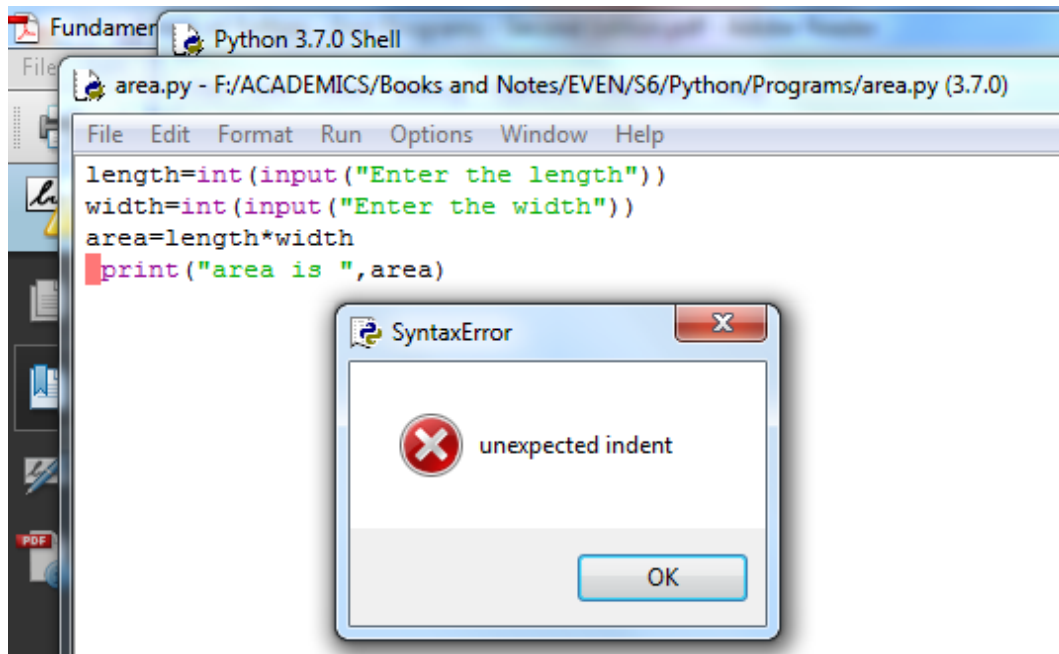
Programmers inevitably make typographical errors when editing programs, and the Python interpreter will nearly always detect them.

Such errors are called syntax errors.

The term *syntax* refers to the rules for forming sentences in a language.

When Python encounters a syntax error in a program, it halts execution with an error message.

```
>>> length = int(input("Enter the length: "))
Enter the length: 44
>>> print(lenth)
Traceback (most recent call last):
  File "<pysHELL#1>", line 1, in <module>
NameError: name 'lenth' is not defined
```

Indentation is significant in Python code.

Each line of code entered at a shell prompt or in a script must begin in the leftmost column, with no leading spaces.

The only exception to this rule occurs in control statements and definitions, where nested statements must be indented one or more spaces.

The Software Development Process

Computer scientists refer to the process of planning and organizing a program as software development.

There are several approaches to software development.

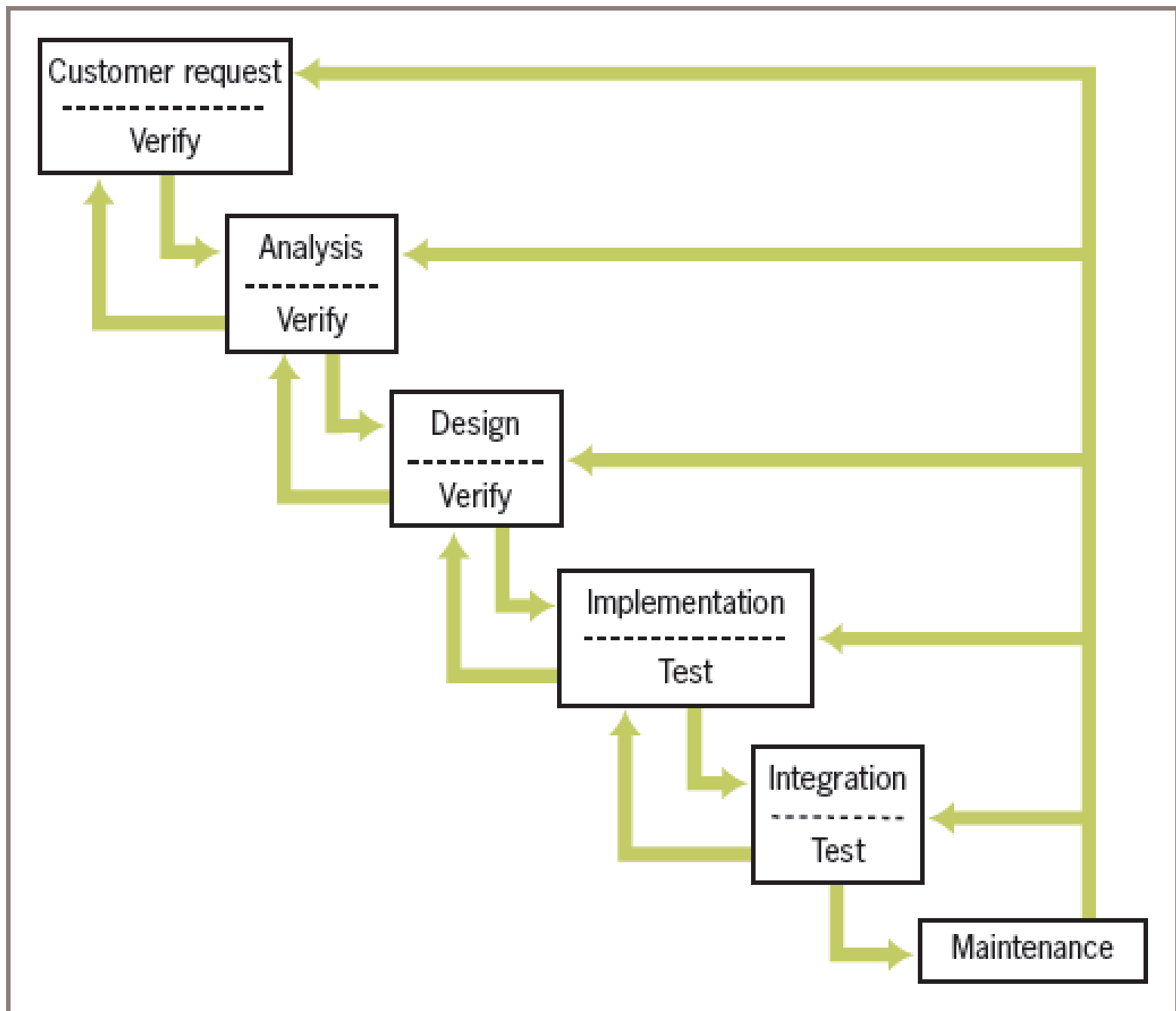
One version is known as the waterfall model.

Results of each phase flow down to the next.

A mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase.

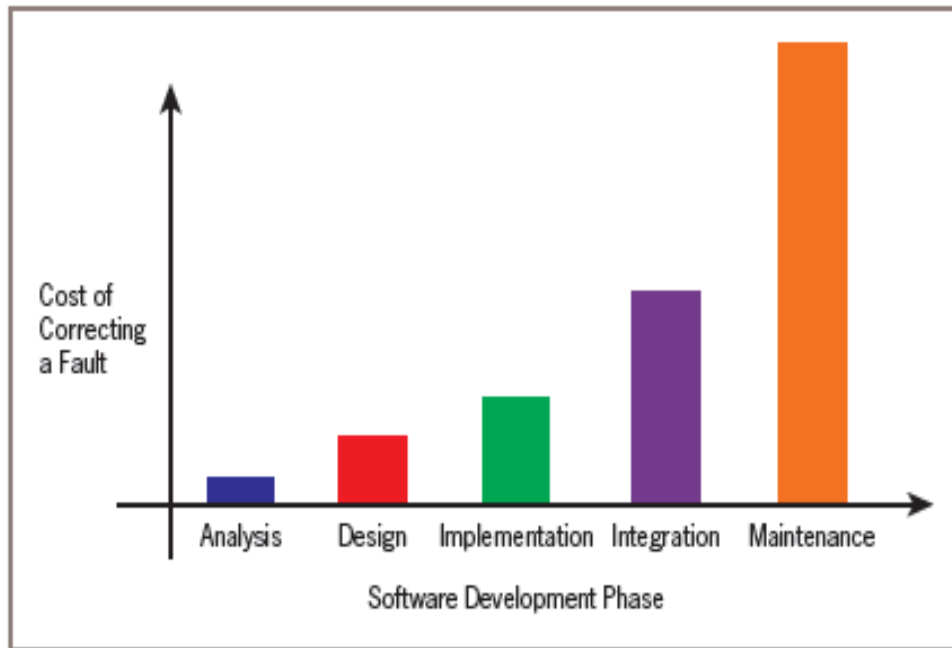
Modifications made during maintenance also require backing up to earlier phases.

Taken together, these phases are also called the **software development life cycle**.



The waterfall model consists of several phases:

1. **Customer request**—In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution. This step is also called the user requirements phase.
2. **Analysis**—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.
3. **Design**—The programmers determine how the program will do its task.
4. **Implementation**—The programmers write the program. This step is also called the coding phase.
5. **Integration**—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.
6. **Maintenance**—Programs usually have a long life; a life span of 5 to 15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made.



Scrutinize the outputs of each phase carefully.

Mistakes found early are much less expensive to correct than those found late.

Maintenance is the most expensive part of software development.

The cost of maintenance can be reduced by careful analysis, design, and implementation.

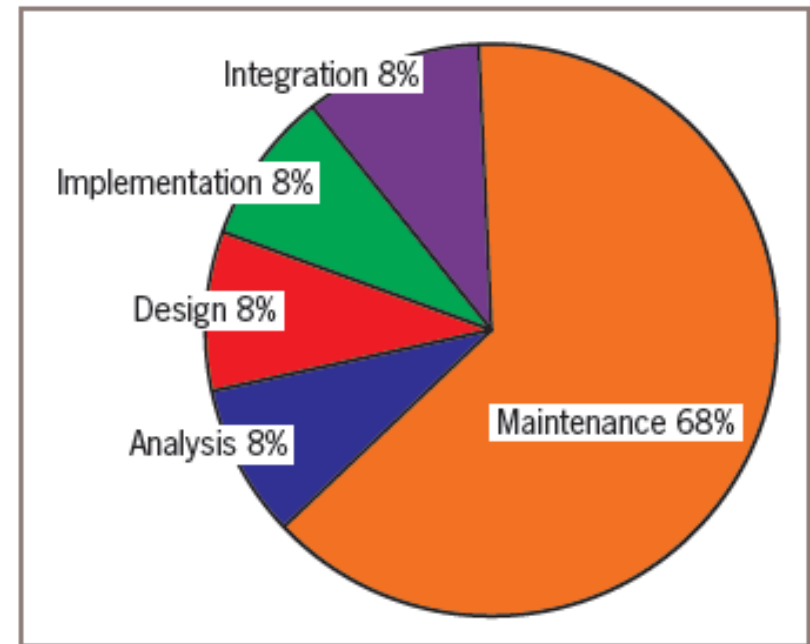


Figure 2-3 Percentage of total cost incurred in each phase of the development process

Case Study: Income Tax Calculator

Each year, nearly everyone with an income faces the unpleasant task of computing his or her income tax return.

Request

The customer requests a program that computes a person's income tax.

Analysis

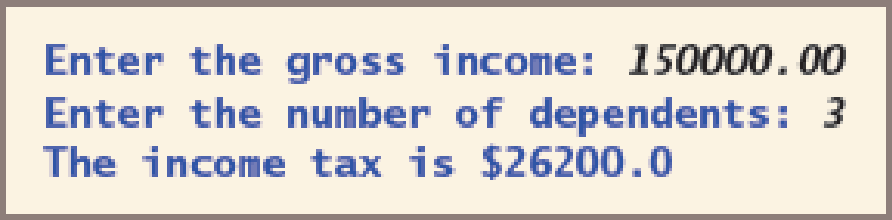
Analysis often requires the programmer to learn some things about the problem domain, in this case, the relevant tax law.

- All taxpayers are charged a flat tax rate of 20%.
- All taxpayers are allowed a \$10,000 standard deduction.
- For each dependent, a taxpayer is allowed an additional \$3,000 deduction.
- Gross income must be entered to the nearest penny.
- The income tax is expressed as a decimal number.

Another part of analysis determines what information the user will have to provide.

In this case, the user inputs are gross income and number of dependents.

The program calculates the income tax based on the inputs and the tax law and then displays the income tax.



```
Enter the gross income: 150000.00  
Enter the number of dependents: 3  
The income tax is $26200.0
```

Figure 2-4 The user interface for the income tax calculator

Design

Describe how the program is going to do it.

This usually involves writing an algorithm.

Algorithms are more often written in a somewhat stylized version of English called **pseudo code**.

```
Input the gross income and number of dependents
Compute the taxable income using the formula
Taxable income = gross income - 10000 - (3000 * number of dependents)
Compute the income tax using the formula
Tax = taxable income * 0.20
Print the tax
```


Implementation (Coding)

Given the preceding pseudo code, an experienced programmer would now find it easy to write the corresponding Python program.

For a beginner, on the other hand, writing the code can be the most difficult part of the process.

```
"""
Program: taxform.py
Author: Ken Lambert
Compute a person's income tax.
1. Significant constants
    tax rate
    standard deduction
    deduction per dependent
2. The inputs are
    gross income
    number of dependents
3. Computations:
    taxable income = gross income - the standard
                    deduction - a deduction for each dependent
    income tax = is a fixed percentage of the taxable income
4. The outputs are
    the income tax
"""
```

```
# Initialize the constants
```

```
TAX_RATE = 0.20
```

```
STANDARD_DEDUCTION = 10000.0
```

```
DEPENDENT_DEDUCTION = 3000.0
```

```
# Request the inputs
```

```
grossIncome = float(input("Enter the gross income: "))
```

```
numDependents = int(input("Enter the number of dependents: "))
```

```
# Compute the income tax
```

```
taxableIncome = grossIncome - STANDARD_DEDUCTION - \
    DEPENDENT_DEDUCTION * numDependents
```

```
incomeTax = taxableIncome * TAX_RATE
```

```
# Display the income tax
```

```
print("The income tax is $" + str(incomeTax))
```

Testing

If there are no syntax errors, enter a set of inputs and view the results.

However, a single run without syntax errors and with correct outputs provides just a slight indication of a program's correctness.

Only thorough testing can build confidence that a program is working correctly.

Testing is a deliberate process that requires some planning and discipline on the programmer's part.

Testing can be performed easily from an IDLE window.

The programmer just loads the program repeatedly into the shell and enters different sets of inputs.

The real challenge is coming up with sets of inputs that can reveal an error.

An error at this point, also called a **logic error or a design error, is an unexpected output.**

A correct program produces the expected output for any legitimate input.

The tax calculator's analysis does not provide a specification of what inputs are legitimate, but common sense indicates that they would be numbers greater than or equal to 0.

Number of Dependents	Gross Income	Expected Tax
0	10000	0
1	10000	-600
2	10000	-1200
0	20000	2000
1	20000	1400
2	20000	800

Table 2-1 The test suite for the tax calculator program

Data Types, Strings, Assignment, and Comments

A **data type** consists of a **set of values** and a **set of operations** that can be performed on those values.

A **literal** is the way a **value of a data type** looks to a programmer.

The programmer can use a literal in a program to mention a data value.

Type of Data	Python Type Name	Example Literals
Integers	<code>int</code>	<code>-1, 0, 1, 2</code>
Real numbers	<code>float</code>	<code>-0.55, .3333, 3.14, 6.0</code>
Character strings	<code>str</code>	<code>"Hi", "", 'A', "66"</code>

Literals for some Python data types

String Literals

A string literal is a sequence of characters enclosed in single or double quotation marks.

```
>>> 'Hello there!'
'Hello there!'
>>> "Hello there!"
'Hello there!'
>>> ''
''
>>> '''
'''
```

empty string

Empty string is different from a string that contains a single blank space character, " ".

Escape Sequences

Escape sequences are the way Python expresses special characters, such as the tab, the newline, and the backspace

Escape Sequence	Meaning
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\\</code>	The <code>\</code> character
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark

String Concatenation

Two or more strings to form a new string using the concatenation operator `+`.

```
>>> "Hi " + "there, " + "Ken!"  
'Hi there, Ken!'
```

The `*` operator allows you to build a string by repeating another string a given number of times.

The left operand is a string, and the right operand is an integer

```
>>> " " * 10 + "Python"  
'          Python'
```


Variables and the Assignment Statement

A variable associates a name with a value

- **Reserved words cannot be used for variable names.**
- **Variable name must begin with either a letter or an underscore (_)**
- **Can contain any number of letters, digits, or other underscores.**
- **Python variable names are case sensitive.**

Variable names that consist of more than one word, it's common to begin each word in the variable name (except for the first one) with an uppercase letter.

Variables receive their initial values and can be reset to new values with an assignment statement.

The simplest form of an assignment statement is the following:

variable name = expression

The Python interpreter first evaluates the expression on the right side of the assignment symbol and then binds the variable name on the left side to this value.

When this happens to the variable name for the first time, it is called **defining or initializing** the variable.

The = symbol means **assignment, not equality**.

After the initialization of a variable, **subsequent uses** of the variable name in expressions are known as **variable references**.

When the interpreter encounters a variable reference in any expression, it looks up the associated value.

If a name is not yet bound to a value when it is referenced, Python signals an error.

Variables serve two important purposes in a program.

1. They help the programmer **keep track of data** that change over time.
2. They also allow the programmer to refer to a complex **piece of information** with a **simple name**.

The wise programmer selects names that inform the human reader about the purpose of the data.

This, in turn, makes the program easier to maintain and troubleshoot.

rate, initialAmount, currentBalance, and interest

Program Comments and Docstrings

A comment is a piece of program text that the computer ignores but that provides useful documentation to programmers.

These comments begin with the # symbol and extend to the end of a line.

An end-of-line comment might explain the purpose of a variable or the strategy used by a piece of code, if it is not already obvious.

```
>>> RATE = 0.85 # Conversion rate for Canadian to US dollars
```

The author of a program can include his or her name and a brief statement about the program's purpose at the beginning of the program file.

This type of comment, called a docstring.
(It is a multiline comment)

```
"""
```

```
Program: circle.py
```

```
Author: Ken Lambert
```

```
Last date modified: 10/10/17
```

```
The purpose of this program is to compute the area of a  
circle. The input is an integer or floating-point number  
representing the radius of the circle. The output is a  
floating-point number labelled as the area of the circle.
```

```
"""
```

Numeric Data Types and Character Sets

Integers

The integers include 0, the positive whole numbers, and the negative whole numbers.

Integer literals are written without commas, and a leading negative sign indicates a negative value.

Although the range of integers is infinite, a real computer's memory places a limit on the magnitude of the largest positive and negative integers.

The most common implementation of the int data type in many programming languages consists of the integers from -2^{31} to $2^{31} - 1$).

However, the magnitude of a Python integer is much larger and is **limited only by the memory of computer.**

Floating-Point Numbers

A real number in mathematics, such as the value of π (3.1416...), consists of a whole number, a decimal point, and a fractional part.

Real numbers have **infinite precision**, which means that the digits in the fractional part can continue forever.

Like the integers, real numbers also have an infinite range.

However, because a computer's memory is not infinitely large, a computer's memory limits not only the range but also the precision that can be represented for real numbers.

Python uses **floating-point numbers to represent real** numbers.

Values of the most common implementation of Python's float type range from approximately -10^{308} to 10^{308} and have 16 digits of precision.

Character Sets

Some programming languages use different data types for strings and individual characters.

In Python, character literals look just like string literals and are of the string type.

'H' as a character and "Hi!" as a string

They are both technically Python strings

All data and instructions in a program are translated to binary numbers before being run on a real computer.

To support this translation, the characters in a string each map to an integer value.

This mapping is defined in character sets

ASCII set and the Unicode set.

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	\$	%	&	`
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	P	q	r	S	t	u	v	w
12	X	y	z	{		}	~	DEL		

The original ASCII character set

The digits in the left column represent the leftmost digits of an ASCII code, and the digits in the top row are the rightmost digits.

Thus, the ASCII code of the character 'R' at row 8, column 2 is 82.

Python's `ord` and `chr` functions convert characters to their numeric ASCII codes and back again.

```
>>> ord('a')  
97  
>>> ord('A')  
65  
>>> chr(65)  
'A'  
>>> chr(66)  
'B'
```

Expressions

Expressions provide an easy way to perform operations on data values to produce other data values

Arithmetic Expressions

An arithmetic expression consists of operands and operators combined.

Operator	Meaning	Syntax
-	Negation	-a
**	Exponentiation	a ** b
*	Multiplication	a * b
/	Division	a / b
//	Quotient	a // b
%	Remainder or modulus	a % b
+	Addition	a + b
-	Subtraction	a - b

Table 2-6

Arithmetic operators

Source: GETKR

Binary operators are placed between their operands : $a * b$

Unary operators are placed before their operands : $-a$

Precedence rules

Exponentiation has the highest precedence and is evaluated first.

Unary negation is evaluated next, before multiplication, division, and remainder.

Multiplication, division, and remainder are evaluated before addition and subtraction.

Addition and subtraction are evaluated before assignment.

Operations of equal precedence are left associative.

Exponentiation and assignment operations are right associative,

(so consecutive instances of these are evaluated from right to left)

Parentheses to change the order of evaluation.

Expression	Evaluation	Value
$5 + 3 * 2$	$5 + 6$	11
$(5 + 3) * 2$	$8 * 2$	16
$6 \% 2$	0	0
$2 * 3 ** 2$	$2 * 9$	18
$-3 ** 2$	$-(3 ** 2)$	-9
$(3) ** 2$	9	9
$2 ** 3 ** 2$	$2 ** 9$	512
$(2 ** 3) ** 2$	$8 ** 2$	64
$45 / 0$	Error: cannot divide by 0	
$45 \% 0$	Error: cannot divide by 0	

Table 2-7 Some arithmetic expressions and their values

- **Syntax** is the **set of rules** for constructing well-formed expressions or sentences in a language.
- **Semantics** is the set of rules that allow an agent to interpret the **meaning** of those expressions or sentences.
- A computer generates a **syntax error** when an expression or sentence is **not well formed**.
- A **semantic error** is detected when the action that an **expression** described, **cannot be carried out**, even though that expression is syntactically correct.
- Although the expressions $45 / 0$ and $45 \% 0$ are syntactically correct, they are meaningless, because a computing agent cannot carry them out.

If **operands** of an arithmetic expression are of **the same numeric type** (int or float), the **resulting value is also of that type**.

When each operand is of a **different type**, the resulting value is of the more **general type**.

Float type is more general than the int type.

The quotient operator `//` produces an integer quotient, whereas the exact division operator `/` always produces a float.

Thus, `3 // 4` produces 0, whereas `3 / 4` produces .75.

When an expression becomes long or complex, you can move to a new line by placing a backslash character `\` at the end of the current line.

```
>>> 3 + 4 * \  
2 ** 5  
131
```

Mixed-Mode Arithmetic and Type Conversions

Performing calculations involving both integers and floating-point numbers is called **mixed-mode arithmetic**.

```
>>> 3.14 * 3 ** 2  
28.26
```

In a binary operation on operands of different numeric types, the **less general type (int)** is temporarily and automatically **converted** to the **more general type (float)** before the operation is performed.

Thus, in the example expression, the value **9** is converted to **9.0** before the multiplication.

Type conversion function

A type conversion function is a function with the same name as the data type to which it converts.

```
>>> radius = input("Enter the radius: ")
Enter the radius: 3.2
>>> radius
'3.2'
>>> float(radius)
3.2
>>> float(radius) ** 2 * 3.14
32.153600000000004
```

int function converts a **float** to an **int** by truncation, not by rounding to the nearest **whole number**.

Truncation simply chops off the number's fractional part.

The **round** function rounds a **float** to the nearest **int**

```
>>> int(6.75)
```

```
6
```

```
>>> round(6.75)
```

```
7
```

Conversion Function	Example Use	Value Returned
<code>int(<a number or a string>)</code>	<code>int(3.77)</code>	3
	<code>int("33")</code>	33
<code>float(<a number or a string>)</code>	<code>float(22)</code>	22.0
<code>str(<any value>)</code>	<code>str(99)</code>	'99'

```
>>> profit = 1000.55
>>> print('$' + profit)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

```
>>> print('$' + str(profit))
$1000.55
```

Construction of strings from numbers and other strings.

Python is a **strongly typed programming language**.

The **interpreter checks data types** of all operands before operators are applied to those operands.

If the type of an operand is not appropriate, the interpreter halts execution with an error message.

Functions and Modules

Python includes many useful functions, which are organized in libraries of code called **modules**.

Calling Functions: Arguments and Return Values

A function is a **chunk of code** that can be **called by name** to perform a task.

Functions often require **arguments**, that is, **specific data values**, to **perform their tasks**.

Names that refer to arguments are also known as **parameters**.

When a function completes its task (which is usually some kind of computation), the **function may send a result back to the part of the program that called that function** in the first place.

The **process of sending a result back** to another part of a program is known as **returning a value**.

Function call `round(6.5)` is the value 6.5, and the value returned is 7.

Function call `abs(4 – 5)` first evaluates the expression `4 – 5` and then passes the result, `-1`, to `abs`. Finally, `abs` returns 1.

Some functions have only **optional arguments**, some have **required arguments**, and some have **both** required and optional arguments.

When called with just one argument, the `round` function exhibits its default behaviour, which is to return the nearest whole number with a fractional part of 0.

`round(7.563, 2)` returns 7.56

Python's help function displays information about round, as follows:

```
>>> help(round)
```

```
Help on built-in function round in module builtin:
```

```
round(...)
```

```
round(number[, ndigits]) -> floating point number
```

```
Round a number to a given precision in decimal digits (default 0 digits).  
This returns an int when called with one argument, otherwise the same type as  
number, ndigits may be negative.
```

The math Module

Functions and other resources are coded in components called **modules**.

Functions like **abs** and **round** from the **__builtin__** module are always available for use, whereas the programmer must **explicitly import** other functions from the modules where they are defined.

The **math module** includes several functions that perform basic mathematical operations.

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'acos', 'acosh', 'asin',
 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 'fsun', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
 'trunc']
```

To use a resource from a module, write the name of a module as a qualifier, followed by a dot (.) and the name of the resource.

```
>>> math.pi
3.1415926535897931
>>> math.sqrt(2)
1.4142135623730951
```

```
>>> help(math.cos)
Help on built-in function cos in module math:
cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

To use only a couple of a module's resources frequently, qualifier can be avoided by importing the individual resources

```
>>> from math import pi, sqrt  
>>> print(pi, sqrt(2))  
3.14159265359 1.41421356237
```

The statement **from math import *** would import all of the math module's resources.

Program Format and Structure

- Start with an introductory comment stating the author's name, the purpose of the program, and other relevant information.

This information should be in the form of a docstring.

- Then, include statements that do the following:

- ☐ **Import any modules needed by the program.**
- ☐ **Initialize important variables, suitably commented.**
- ☐ **Prompt the user for input data and save the input data in variables.**
- ☐ **Process the inputs to produce the results.**
- ☐ **Display the results.**

Control statements

Loops and Selection Statements

Definite Iteration: The for Loop

Control statements with repetition statements, also known as **loops**, which repeat an action.

Each repetition of the action is known as a **pass** or an **iteration**.
There are two types of loops—

Those that repeat an action **a predefined number** of times (**definite iteration**)

Those that perform the action **until the program** determines that it needs to stop (**indefinite iteration**).

```
>>> for eachPass in range(4):  
    print("It's alive!", end = " ")  
It's alive! It's alive! It's alive! It's alive!
```

This loop repeatedly calls one function—the print function.

The constant 4 on the first line tells the loop how many times to call this function.

To print 10 or 100 exclamations, change the 4 to 10 or to 100.

```
for <variable> in range(<an integer expression>):  
    <statement-1>  
    .  
    .  
    <statement-n>
```

The **first line** of code in a loop is sometimes called the **loop header**.

The only relevant information in the header is the integer expression, which denotes the number of iterations that the loop performs.

The **colon (:) ends the loop header**.

The loop body comprises the statements in the remaining lines of code, below the header.

These statements are executed in sequence on each pass through the loop.

The statements in the loop body ***must be indented and aligned in the same column***.

Count-Controlled Loops

When Python executes the type of for loop discussed above, it counts from 0 to the value of the header's integer expression minus 1.

On each pass through the loop, the header's variable is bound to the current value of this count.

```
>>> for count in range(4):  
        print(count, end = " ")  
0 1 2 3
```

Loops that count through a range of numbers are also called count-controlled loops.

The value of the count on each pass is often used in computations.

```
>>> product = 1  
>>> for count in range(4):  
        product = product * (count + 1)  
>>> product  
24
```

To count from an **explicit lower bound**, the programmer can supply a **second integer expression** in the loop header.

When two arguments are supplied to range, the count ranges from the first argument to the second argument minus 1.

```
>>> product = 1
>>> for count in range(1, 5):
        product = product * count
>>> product
24
```

Second argument of range, which should specify an integer greater by 1 than the desired upper bound of the count.

```
for <variable> in range(<lower bound>, <upper bound + 1>):
    <loop body>
```

```
>>> lower = int(input("Enter the lower bound: "))
Enter the lower bound: 1
>>> upper = int(input("Enter the upper bound: "))
Enter the upper bound: 10
>>> theSum = 0
>>> for number in range(lower, upper + 1):
        theSum = theSum + number
>>> theSum
55
```

Augmented Assignment

The assignment symbol can be combined with the arithmetic and concatenation operators to provide **augmented assignment operations**.

```
a = 17
s = "hi"
a += 3      # Equivalent to a = a + 3
a -= 3      # Equivalent to a = a - 3
a *= 3      # Equivalent to a = a * 3
a /= 3      # Equivalent to a = a / 3
a %= 3      # Equivalent to a = a % 3
s += " there" # Equivalent to s = s + " there"
```

All these examples have the format

<variable> <operator>= <expression>

which is equivalent to

<variable> = <variable> <operator> <expression>

Loop Errors: Off-by-One Error

The loop fails to perform the expected number of iterations.

Because this number is typically off by one, the error is called an **off-by-one error**.

For the most part, off-by-one errors result when the programmer incorrectly specifies the upper bound of the loop.

The programmer might intend the following loop to count from 1 through 4, but it counts from 1 through 3:

```
# Count from 1 through 4, we think
>>> for count in range(1,4):
    print(count)
1
2
3
```


Traversing the Contents of a Data Sequence

The loop itself visits each number in a sequence of numbers generated by the **range** function.

```
>>> list(range(4))  
[0, 1, 2, 3]  
>>> list(range(1, 5))  
[1, 2, 3, 4]
```

The sequence of numbers generated by the function range is fed to Python's list function, which returns a special type of sequence called a **list**.

The values contained in any sequence can be visited by running a for loop

```
for <variable> in <sequence>:  
    <do something with variable>
```

```
>>> for number in [6, 4, 8]:  
    print(number, end = " ")  
6 4 8  
>>> for character in "Hi there!":  
    print(character, end = " ")  
H i   t h e r e !
```

Specifying the Steps in the Range

A variant of Python's range function expects a third argument that allows you to nicely skip some numbers.

The third argument specifies a **step value, or the interval between the numbers used in the range,**

```
>>> list(range(1, 6, 1))    # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))    # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))    # Use every third number
[1, 4]
```

```
>>> theSum = 0
>>> for count in range(2, 11, 2):
        theSum += count
>>> theSum
30
```

Loops That Count Down

From the upper bound down to the lower bound.

```
>>> for count in range(10, 0, -1):  
    print(count, end = " ")  
10 9 8 7 6 5 4 3 2 1  
>>> list(range(10, 0, -1))  
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

When the step argument is a negative number, the range function generates a sequence of numbers from the first argument down to the second argument plus 1.

Thus, in this case, the first argument should express the upper bound, and the second argument should express the lower bound **minus 1**.

Formatting Text for Output

Many data-processing applications require output that has a **tabular format**, like that used in spreadsheets or tables of numeric data.

In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified.

A column of data is left-justified if its values are vertically aligned beginning with their leftmost characters.

A column of data is right-justified if its values are vertically aligned beginning with their rightmost characters.

To **maintain the margins** between columns of data, **left-justification requires the addition of spaces to the right of the datum, whereas right-justification requires adding spaces to the left of the datum.**

The **total number of data characters** and **additional spaces** for a given datum in a formatted string is called its **field width**.

The print function automatically begins printing an output datum in the first available column.

The next example, which displays the exponents 7 through 10 and the values of 10^7 through 10^{10} , shows the format of two columns produced by the print function:

```
>>> for exponent in range(7, 11):  
        print(exponent, 10 ** exponent)  
7 10000000  
8 100000000  
9 1000000000  
10 10000000000
```

when the exponent reaches 10, the output of the second column shifts over by a space and looks ragged.

Python includes a general formatting mechanism that allows the programmer to specify field widths for different types of data.

```
>>> "%6s" % "four"          # Right justify  
'  four'  
>>> "%-6s" % "four"         # Left justify  
'four '
```

The simplest form of this operation is the following:

<format string> % <datum>

contains a **format string**, the **format operator %**, and a **single data value** to be formatted.

%<field width>s

When the field width is positive, the datum is right-justified;

When the field width is negative, the datum is left-justified.

If the field width is less than or equal to the datum's print length in characters, no justification is added.

The % operator works with this information to build and return a formatted string

To format integers, use the letter d instead of s.

```
>>> for exponent in range(7, 11):
        print("%-3d%12d" % (exponent, 10 ** exponent))
7      10000000
8      100000000
9      1000000000
10     10000000000
```

The format information for a data value of type **float** has the form
%<field width>.<precision>f
where .<precision> is optional.

```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
```

```
>>> "%6.3f" % 3.14
' 3.140'
```

Python adds a digit of precision to the string and pads it with a space to the left to achieve the field width of 6. This width includes the place occupied by the decimal point.

Selection structure

Selection: if and if-else Statements

Instead of moving straight ahead to execute the next instruction, the computer might be faced with two alternative courses of action.

The computer must pause to examine or test a condition.

If the condition is true, the computer executes the first alternative action and skips the second alternative.

If the condition is false, the computer skips the first alternative action and executes the second alternative.

if-else Statements

The if-else statement is the most common type of selection statement.

It is also called a **two-way selection statement**, because it directs the computer to make a choice between two alternative courses of action.

```
if <condition>:  
    <sequence of statements-1>  
else:  
    <sequence of statements-2>
```

Python syntax for the if-else statement

```
import math  
area = float(input("Enter the area: "))  
if area > 0:  
    radius = math.sqrt(area / math.pi)  
    print("The radius is", radius)  
else:  
    print("Error: the area must be a positive number")
```

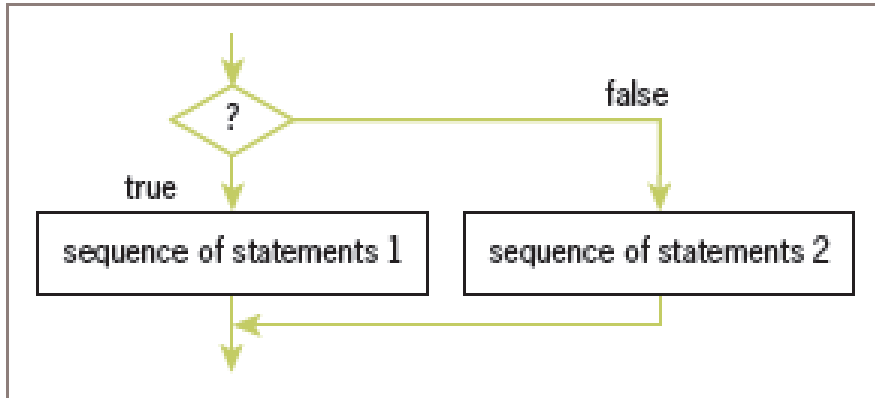


Figure 3-2 The semantics of the `if-else` statement

Each sequence *must be indented at least one space* beyond the symbols `if` and `else`.

Note the use of the colon (`:`) following the condition and the word `else`.

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
print("Maximum:", max(first, second))
print("Minimum:", min(first, second))
```

One-Way Selection Statements

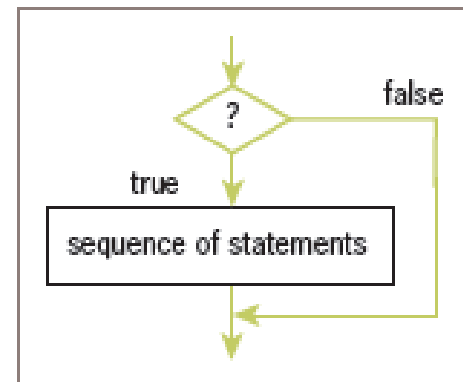
The simplest form of selection is the if statement.

This type of control statement is also called a **one-way selection statement**, because it consists of a condition and just a single sequence of statements.

If the condition is True, the sequence of statements is run.

Otherwise, control proceeds to the next statement following the entire selection statement.

```
if <condition>:  
    <sequence of statements>
```



Multi-Way if Statements

For testing several conditions that entail more than two alternative courses of action.

Letter Grade	Range of Numeric Grades
A	All grades above 89
B	All grades above 79 and below 90
C	All grades above 69 and below 80
F	All grades below 70

```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

```
if <condition-1>:  
    <sequence of statements-1>  
elif <condition-n>:  
    <sequence of statements-n>  
else:  
    <default sequence of statements>
```

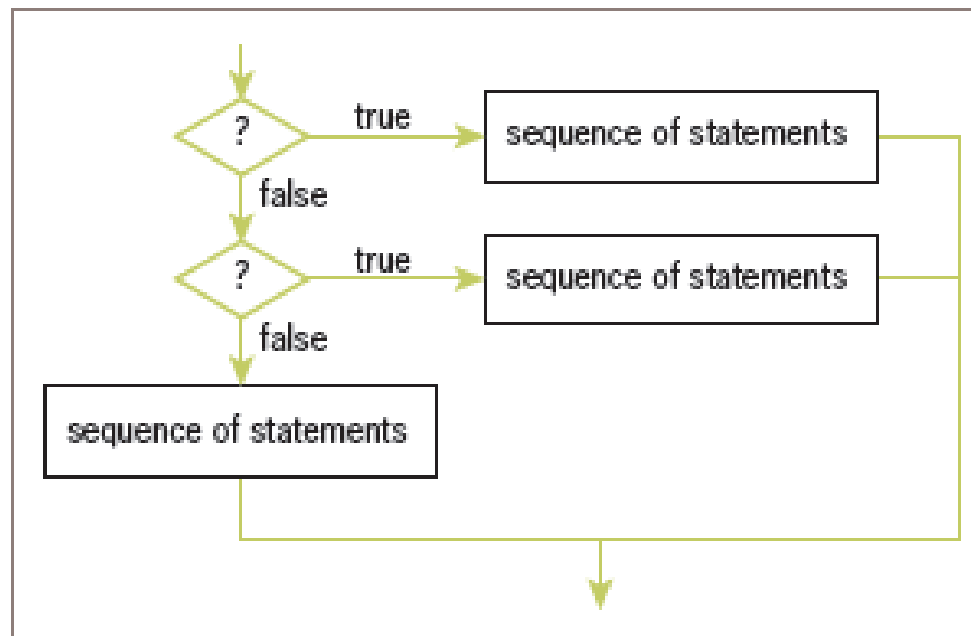


Figure 3-4 The semantics of the multi-way if statement

The Boolean Type, Comparisons, and Boolean Expressions

The Boolean data type consists of only two data values
— true and false.

Simple Boolean expressions consist of the Boolean values True or False, variables bound to those values, function calls that return Boolean values, or comparisons.

The condition in a selection statement often takes the form of a comparison.

Comparison Operator	Meaning
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

```
>>> 4 == 4
```

```
True
```

```
>>> 4 != 4
```

```
False
```

```
>>> 4 < 5
```

```
True
```

```
>>> 4 >= 3
```

```
True
```

```
>>> "A" < "B"
```

```
True
```

Python includes all three Boolean or logical operators.

and, or, and not.

Both the and operator and the or operator expect two operands.

The and operator returns True if and only if both of its operands are true, and returns False otherwise.

The or operator returns False if and only if both of its operands are false, and returns True otherwise.

The not operator expects a single operand and returns its **logical negation, True, if it's false, and False if it's true.**

Logical Operators and Compound Boolean Expressions

Two conditions can be combined in a Boolean expression that uses the logical operator **Or**.

The result is a compound Boolean expression

```
number = int(input("Enter the numeric grade: "))
if number > 100:
    print("Error: grade must be between 100 and 0")
elif number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

```
number = int(input("Enter the numeric grade: "))
if number > 100 or number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

```
number = int(input("Enter the numeric grade: "))
if number >= 0 and number <= 100:
    # The code to compute and print the result goes here
else:
    print("Error: grade must be between 100 and 0")
```

Type of Operator	Operator Symbol
Exponentiation	**
Arithmetic negation	-
Multiplication, division, remainder	*, /, %
Addition, subtraction	+, -
Comparison	==, !=, <, >, <=, >=
Logical negation	not
Logical conjunction	and
Logical disjunction	or
Assignment	=

Table 3-4 Operator precedence, from highest to lowest

Short-Circuit Evaluation

The Python virtual machine sometimes knows the value of a Boolean expression before it has evaluated all of its operands.

In the expression A and B, if A is false, then so is the expression, and there is no need to evaluate B.

Likewise, in the expression A or B, if A is true, then so is the expression, and again there is no need to evaluate B.

This approach, in which evaluation stops as soon as possible, is called **short-circuit evaluation**.

```
count = int(input("Enter the count: "))
theSum = int(input("Enter the sum: "))
if count > 0 and theSum // count > 10:
    print("average > 10")
else:
    print("count = 0 or average <= 10")
```

If the user enters 0 for the count, the condition contains a potential division by zero; however, because of short-circuit evaluation the division by zero is avoided.

Testing Selection Statements

Selection statements add extra logic to a program, they open the door for extra logic errors.

Thus, take special care when testing programs that contain selection statements.

The first rule of thumb is to make sure that all of the possible **branches or alternatives in a selection statement** are exercised.

This will happen if the test data include values that make each condition true and also each condition false.

After testing all of the actions, also **examine all of the conditions**.

Finally, test conditions that contain **compound Boolean expressions** using data that produce all of the possible combinations of values of the operands.

Conditional Iteration: The while Loop

The **number of iterations** in a loop is **unpredictable** in certain cases.

The loop eventually completes its work, but only when a condition changes.

The loop continues to repeat as long as a condition remains **true**.

This type of process is called **conditional iteration**

The Structure and Behaviour of a while Loop

Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue.

Such a condition is called the loop's **continuation condition**.

If the continuation condition is false, the loop ends.

If the continuation condition is true, the statements within the loop are executed again.

```
while <condition>:  
    <sequence of statements>
```

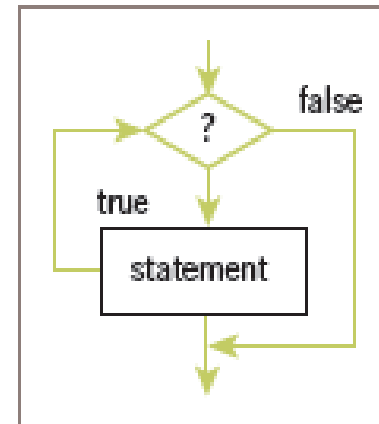


Figure 3-6 The semantics of a while loop

```
theSum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    theSum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", theSum)
```

```
Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```

loop control variable

The while loop is also called an **entry-control loop**

Its condition is tested at the top of the loop.

This implies that the statements within the loop can execute zero or more times.

The while True Loop and the break Statement

The **loop's entry condition** is the Boolean value **True**.

This condition is extremely easy to write and guarantees that the body of the loop will execute at least once.

```
theSum = 0.0
while True:
    data = input("Enter a number or just enter to quit: ")
    if data == "":
        break
    number = float(data)
    theSum += number
print("The sum is", theSum)
```

The break statement will cause an exit from the loop


```
done = False
while not done:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number) # Just echo the valid input
```

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated,

Loop Logic, Errors, and Testing

Errors during testing the while loop

- Incorrectly initialized loop control variable
- Failure to update this variable correctly within the loop
- Failure to test it correctly in the continuation condition.

If one simply forgets to update the control variable, the result is an infinite loop, which does not even qualify as an algorithm!

To halt a loop that appears to be hung during testing, type **Control+c** in the terminal window or in the IDLE shell.

Lazy evaluation

Lazy evaluation is a strategy implemented in some programming languages.

Certain objects are not produced until they are needed.

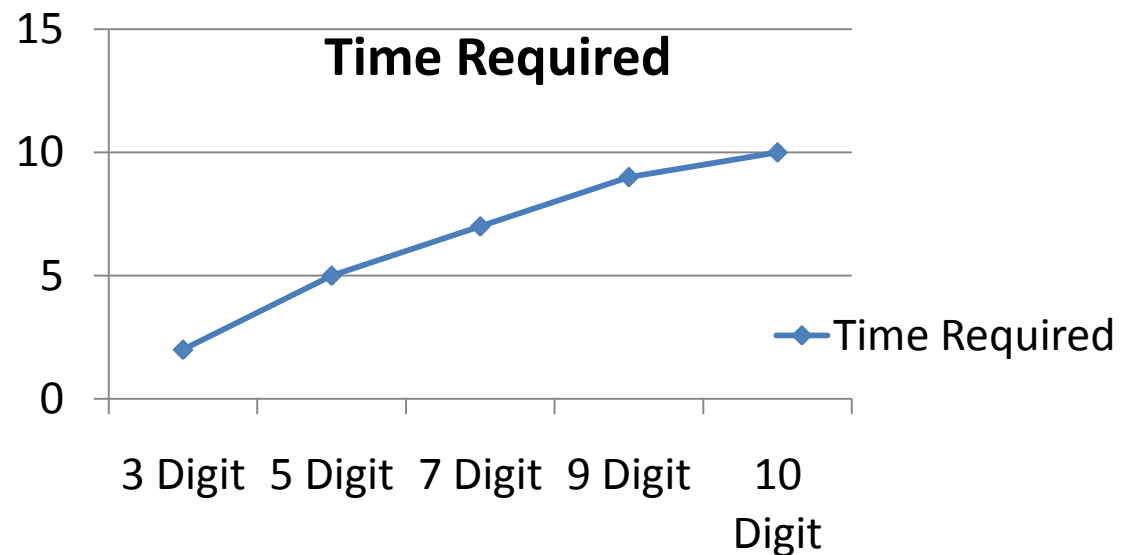
This strategy is often used in conjunction with functions that produce collections of objects.

For efficient memory management and run time.

To iterate through a sequence of objects that can be generated and acted upon one at a time, then **no need to explicitly construct a big container like a list**, only to process the elements one-by-one.

1. Write a program to find the GCD of two numbers.
2. Plot a graph which shows the time required to find the GCD of the following list of numbers.

1. Two 3 digit numbers
2. Two 5 digit numbers.
3. Two 7 digit numbers.
4. Two 9 digit numbers.
5. Two 10 digit numbers.



Assignment: 1

Date: 01/06/2022

End Module 1