INDUSTRIAL PROGRAMMING

PYTHON

Introduction

- **Python programming language was invented by Guido van Rossum (Shedo fan Rossum) in 1990**
- **Guido van Rossum was a big fan of the TV show Monty Python's Flying Circus so he named the new programming language Python**
- **Python is a high-level, general-purpose programming language for solving problems**
- **We can download Python, its documentation and related materials from www.python.org**
- **Python is an interpreted language**

Introduction

- **We can run Python commands and expressions in an interactive programming environment called the Python shell which comes with the Python installation**

  eg:- 3+5

  print(“ Today is Monday”)

Printf function

- **The printf function is used to print data in the output device which is the display**
- **The syntax of print function is print(<expression>)**
- **Python first evaluates the expression and then displays its value**
- **The syntax for print statement with two or more expressions is print(<expression>,…,<expression>)**

  Eg:- print("The area is", area, "square units")
- **The print function displays the values of the expression and then moves the cursor to the next new line**

printf( ) function

- To begin the next output on the same line as the previous one use the expression end=""
- print(<expression>, end="")
- By using the expression end="" the cursor will not move to the next line and will continue in the same line to print the next expression eg:- print("The area is ", end="")

  print(area,end="")

  print(" square units")
- This will produce the out put

  The area is 500 square units

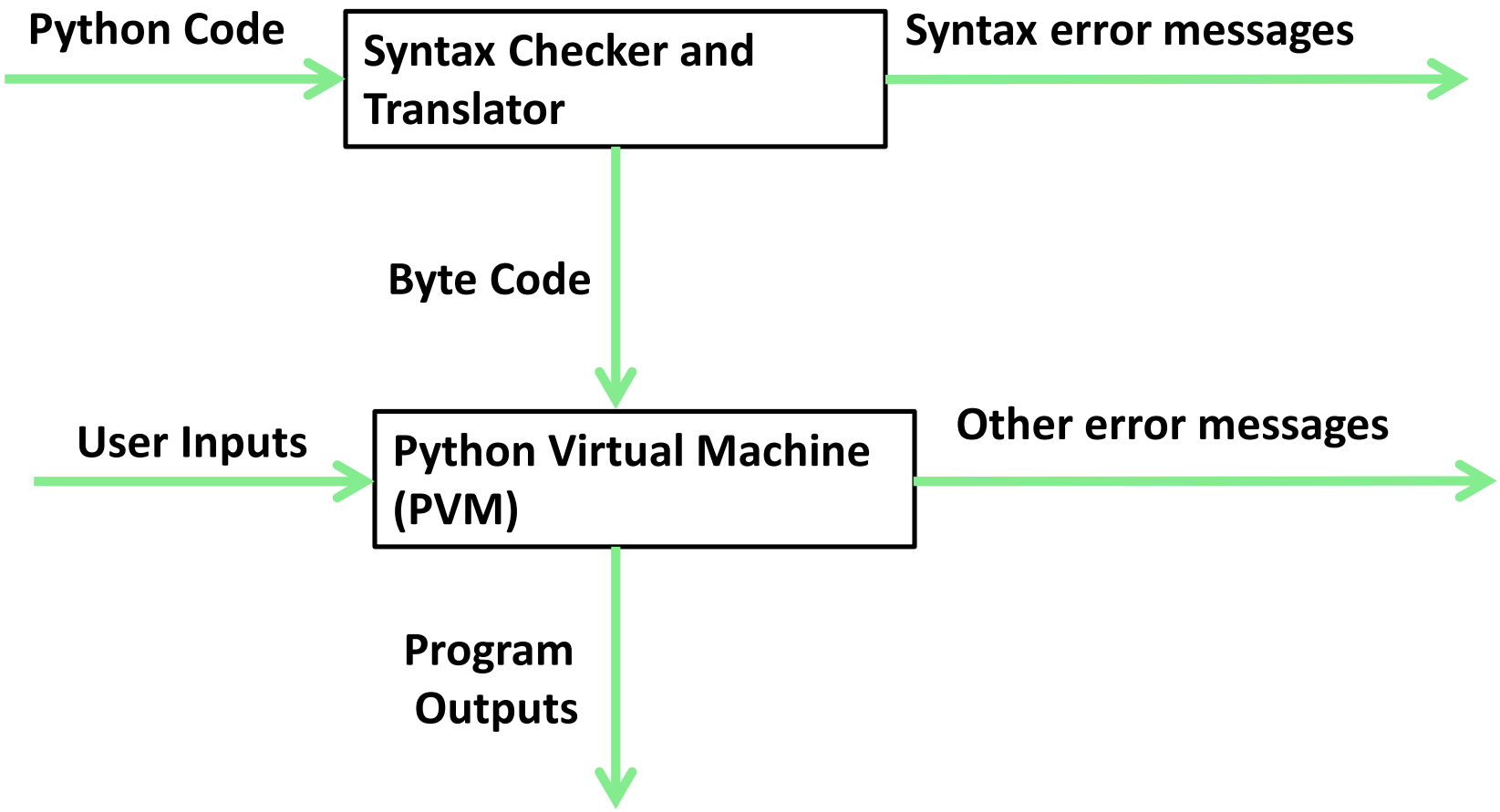
input( ) function

- **The input function accepts the value entered by the user and makes it available for the program**
- **The syntax of input function is**

  variable = input(prompt)

  eg:- x = input('Enter your name:')
- **The input function always returns a string value from the user's keystrokes to the program**
- **We have to convert the string value to integer and float value using int and float functions before using in programs**

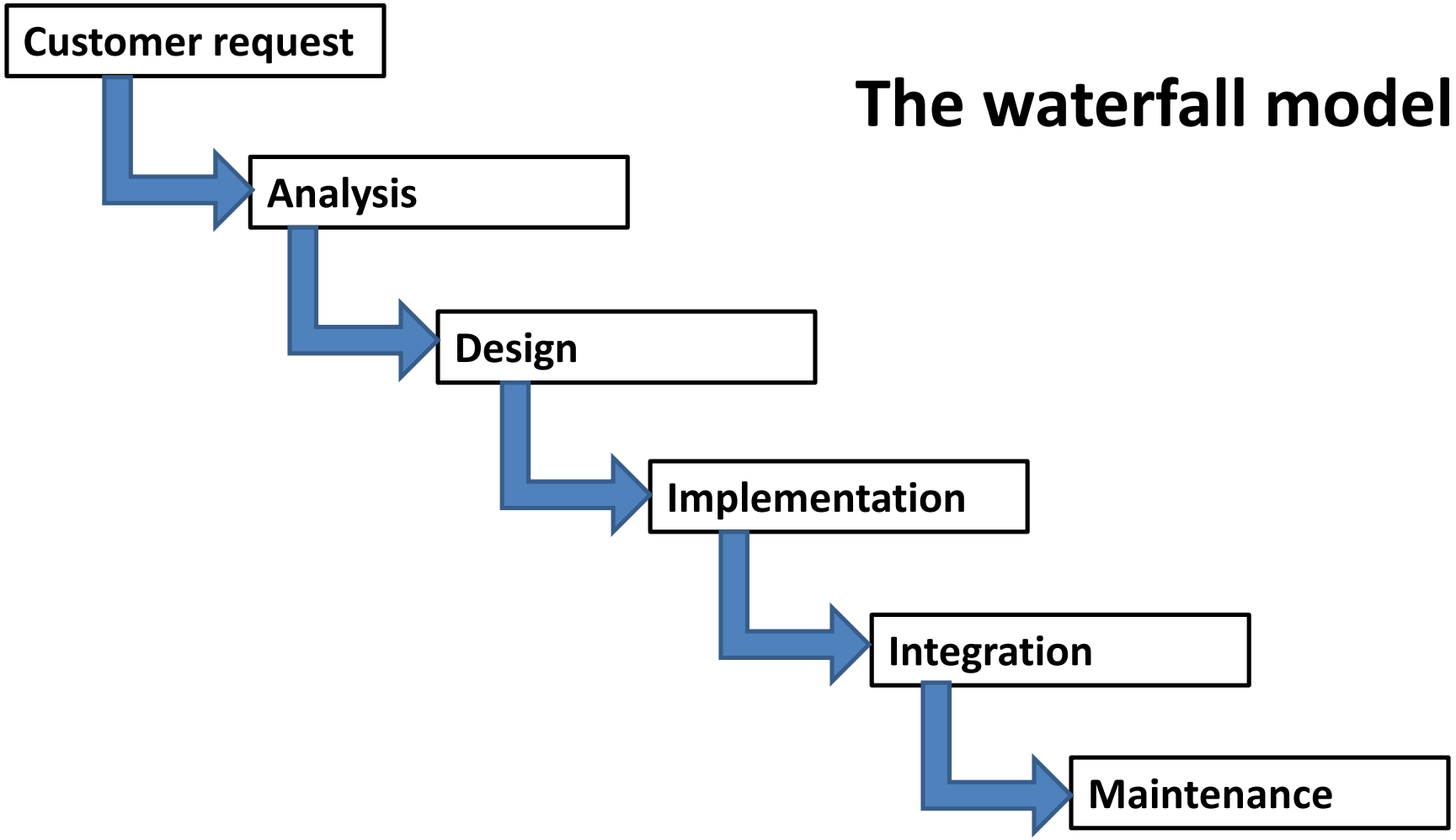
How Python Works



How Python Works

1. **The interpreter reads a Python code and verifies that it is well formed. It check for syntax errors and sent error messages**
2. **If a Python expression is well formed the interpreter then translates it to an equivalent form in a low-level language called byte code**
3. **The byte code is next sent to another software component called the Python virtual machine (PVM) where it is executed**

The Software Development Process



The waterfall model

The waterfall model

- **Customer request or Requirement Phase — In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution**
- **Analysis—The programmers determine what the program will do or it is viewed as a process of clarifying the specifications for the problem**
- **Design—The programmers determine how the program will do its task**
- **Implementation or The coding Phase —The programmers write the program**

The waterfall model

- **Integration— In the integration phase the parts are brought together into a smoothly functioning whole unit**
- **Maintenance—Programs usually have a long lifespan during this time, requirements change, errors are detected, and minor or major modifications are made**
- **This model resembles a waterfall in which the results of each phase flow down to the next phase**
- **Programs rarely work as hoped the first time they are run hence, they should be subjected to extensive and careful testing in Implementation and Integration phases**

Case Study

- **Request - The customer requests a program that computes a person's income tax**
- **Analysis - Analysis often requires the programmer to learn about the problem domain in this case the tax law**

  The Tax Law states that

  All taxpayers are charged a flat tax rate of 20%.

  All taxpayers are allowed a Rs10,000 standard deduction

  For each dependent a taxpayer is allowed an additional Rs 3,000 deduction

Case Study

- **The analysis determines what information the user will have to provide**
- **In this case the user inputs are gross income and number of dependents**
- **Design- The Design phase specify what a program is going to do**
- **The Design involves writing an algorithm**

    1. Input the gross income and number of dependents

    2. Compute the taxable income using the formula

Case Study

3. Taxable income = gross income - 10000 - (3000 * number of dependents)

4. Tax = taxable income * 0.20

5. Print the tax

- The algorithm can be converted to Python code
- Implementation (Coding) – The algorithm is converted to Python code in the Implementation Phase

Case Study

```
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
```

Case Study

```
incomeTax = taxableIncome * TAX_RATE

# Display the income tax

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

- **Testing – In the testing phase the program is run against different sets of input that can reveal errors**
- **In the tax program we try inputs of 0, 1, and 2 for the number of dependents**
- **If the program works correctly with these we can assume that it will work correctly with larger values**

Data Types

Type of Data	Python Type Name	Example Literals
Integers	int	-1, 0, 1, 2
Real Numbers	float	-0.55, .3333, 3.14, 6.0
Character Strings	str	“Hi”, “ “, ‘A’, ‘66’

- **A data type is the kinds of data a variable can hold**
- **The first two data types int and float, are called numeric data types because they represent numbers**
- **A string literal is a sequence of characters enclosed in single or double quotation marks**

String Literal

- **“” – empty string**
- **“ “- single blank space which is different from empty string**
- **“How are you?” – is a string**
- **The print function displays the nested quotation mark**

  print(“Loyd’s Car”)
- **To output a paragraph of text that contains several lines it is more convenient to enclose the entire string literal line breaks and all within three consecutive quotation marks**

  print("""This very long sentence extends all the way to

  the next line.""")

Escape Sequences

Escape Sequence	Meaning
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	The \ character
\’	Single Quotation Mark
\”	Double Quotation Mark

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

String Concatenation

- **Two or more strings can be joined together to form a new string using the concatenation operator +**

```
>>> "Today " + "is " + "Monday "
'Today is Monday '
```

- **The * operator can be used to build a string by repeating another string a given number of times**

```
>>> "python " * 10
'python python python python python python python python python python '
```

Variables

- **A variable associates a name with a value, making it easy to remember and use the value later in a program**
- **There are a few rules to obey while choosing names for variables**

1. **Reserve words cannot be used for variable names like if, def etc.**
2. **A variable name must begin with either a letter or an underscore (_), and can contain any number of letters, digits, or underscores**
3. **Variable name should not begin with a digit**
4. **Python variable names are case sensitive thus, the variable WEIGHT is a different name from the variable weight**

Variables

- **Programmers typically use lowercase letters for variable names but in the case of variable names that consist of more than one word it is common to begin each word in the variable name with an uppercase letter which makes the variable name easier to read**

eg. the name interestRate is slightly easier to read than interestrate

- **Programmers use all uppercase letters for the names of variables that contain values that the program never changes and such variables are known as symbolic constants**

eg. TAX_RATE

Variables

- **Variables receive their initial values and can be reset to new values with an assignment statement**
- **<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**
- **The = symbol means assignment not equality**
- **A good programmer selects the name of variable in such a way that the name gives the information about the contents of the data it holds**

Comments in Python

- **A comment is a piece of program text that the interpreter ignores but that provides useful documentation to programmers**
- **There are two types of comments in Python**
- **A programmer can give a multi-line string about the purpose of the program at the beginning of the program file known as docstring**
- **A multi-line comment can be inserted between three set of double quotes**

“ “ “ Comment “ “ “

Comments in Python

- **An end-of-line comment begin with the # symbol and extend to the end of a line**
- **The end-of-line comment normally explains the purpose of a variable or the strategy used by a piece of code**

```
rate = 85.83  # Conversion rate for US dollar to Indian rupees
```

The ASCII Character Set

- **In the 1960s the original ASCII set encoded each keyboard character and several control characters using the integers from 0 through 127**
- **In the following ASCII chart 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**

  eg:- The ASCII code of the character 'R' at row 8, column 2 is 82
- **In Python *ord* function convert characters to their numeric ASCII codes and *chr* functions convert numeric ASCII codes to their corresponding characters**

The ASCII Character 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	DCI	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 ASCII Character Set

```
>>>ord('a')
97
>>>ord('A')
65
>>>chr(65)
'A'
>>>chr(66)
'B'
>>> chr(ord('A') + 3)
'D'
```

Arithmetic Expressions

Operator	Meaning	Syntax	Precedence
**	Exponentiation	a ** b	5
-	Negation	-a	4
*	Multiplication	a * b	3
/	Division	a / b	3
//	Quotient	a // b	3
%	Remainder or Modulus	a % b	3
+	Addition	a + b	2
-	Subtraction	a - b	2

- An arithmetic expression consists of operands and operators
- If an expression extends more than a line it can be continued to the next line by placing a backslash character \ at the end of the current line

Arithmetic Expressions

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

Arithmetic type Conversion

- **In Python while conducting a mathematical 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**
- **eg:- The value 12 is converted to 12.0 before the multiplication**
- **A type conversion function should be used when working with the input of numbers**
- **The input function returns a string as its value so we have to use the function int or float to convert the string to a number before performing arithmetic operations**

Type conversion

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

- The int function converts a float to an integer
- The round function rounds a float to the nearest integer
- >>> round(6.5)

  6

- >>> round(6.51)

  7

Arithmetic type Conversion

- **>>> profit = 1000.55**
- **>>> print(‘Rs' + profit)**
- **Python does not allow the use of the concatenation operator (+) with a string and a number**
- **To solve this problem the str function is used to convert the value of profit to a string and then concatenate this string to the Rs symbol**

```
print('Rs ' + str(profit))
```

Rs 1000.55

Functions and Modules

- **Python includes many useful functions which are organized in libraries of code called modules**
- **A function is a group of code that can be called by name to perform a task**
- **Arguments or parameters is a kind of variables used to pass data to a function**
- **The process of sending a result back to another part of a program is known as returning a value**
- **The argument in the function call round(6.6) is the value 6.6 and the value returned is 7**

Functions and Modules

- **Instead of a value the argument can be an expression and in case of expression the expression is first evaluated and its value is passed to the function**
- **Eg;- abs(4 - 5) will return 1**
- **Some functions have required arguments and optional arguments also**
- **Eg:- round(6.6) and round(6.476, 2) will give 6.48**
- **With one argument the round function return the nearest float value and with second optional argument which indicates the number of places of precision the function returns the value accordingly**

Functions and Modules

- **To know about a function and its arguments python has a facility called help**
- **Help can be called from the shell prompt**

  Eg:- >>> help(round)
- **Each argument passed to a function has a specific data type**
- **A program that attempts to pass an argument of the wrong data type to a function will generate an error**
- **Functions and other resources are coded in components called modules**

Functions and Modules

- **The programmer must explicitly import functions from the modules where they are defined**
- **The math module includes several functions that perform basic mathematical operations**

```
import math
dir(math)
```

- **This commands in the shell script will display all the functions available in the math module**
- **To use a resource from a module we have to write the name of a module as a qualifier followed by a dot (.) and the name of the resource eg:- math.pi**

Control Statements

- **Computers can be used to select a specific action to perform in a particular situation**
- **Computers are able to repeat a set of actions**
- **Control statements allow the computer to select or repeat an action**
- **In certain cases the computers have to take alternative statements depending on certain conditions**
- **If the condition is true the computer executes the first alternative action and skips the second alternative**

Control Statements

- If the condition is false the computer skips the first alternative action and executes the second alternative
- Selection statements are used to take alternative statements depending on conditions
- Python's comparison operators are

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

If Statement

- **The simplest form of selection statement is the if statement**
- **The if 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**
- **The syntax for the if statement**

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

If-else Statement

- **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 the condition is True one sequence of statements is run otherwise another sequence of statements is executed**
- **Remember the use of colon (:) following the condition and the word else**

If-else Statement

- **The syntax of the if-else statement is**

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

Multi-Way if Statement

- **In certain cases the computer has to choose between a set of conditions in those cases a multi-way selection statement can be used**
- **The multi-way if statement considers each condition until one evaluates to True or they all evaluate to False**
- **When a condition evaluates to True the corresponding action is performed and control skips to the end of the entire selection statement**
- **If no condition evaluates to True then the action after the trailing else is performed**

Multi-Way if Statement

- **The syntax of the Multi-way if statement is**

```
if <condition-1>:
    <sequence of statements-1>
elif <condition-2>:
    <sequence of statements-2>
elif <condition-3>:
    <sequence of statements-3>
 ----------------------
elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

Loop Control Statements

- **Certain tasks are repetitive in nature**
- **Loops can we used to solve tasks which are repetitive in nature**
- **Each repetition of the action is known as a pass or an iteration**
- **There are two types of loops**

1. **definite iteration - those that repeat an action a predefined number of times**
2. **indefinite iteration - those that perform the action until the program determines that it needs to stop**

- **In Python the for loop supports definite iteration**

For Loop

- **Certain tasks are repetitive in nature**
- **Loops can we used to solve tasks which are repetitive in nature**
- **Each repetition of the action is known as a pass or an iteration**
- **There are two types of loops**

1. **definite iteration - those that repeat an action a predefined number of times**
2. **indefinite iteration - those that perform the action until the program determines that it needs to stop**

- **In Python the for loop supports definite iteration**

For Loop

- **The Syntax of for loop is**

  for <variable> in range(<an integer expression>):

  <statement-1>

  <statement-2>

  <statement-3>

  <statement-n>

- **The first line of the syntax is called the loop header**
- **The colon (:) ends the loop header**
- **The loop body must be indented and aligned**

For Loop

- **Eg. The code**

```
for counter in range(4):
    print(str(counter) + ". Python programming is Fun")
```

- **Will produce the output**

```
0. Python programming is Fun
1. Python programming is Fun
2. Python programming is Fun
3. Python programming is Fun
```

- **The counter value changes from 0 to 3 in total counter will have 4 values**

For Loop

- **The for loop has another version where there is a lower bound and upper bound for the range**

for <variable> in range(<<lower bound>, <upper bound>):

<statement-1>

<statement-2>

<statement-3>

<statement-n>

For Loop

```
for counter in range(1,4):
    print(str(counter) + ". Python programming is Fun")
```

- Will produce the output

    1. Python programming is Fun
    2. Python programming is Fun
    3. Python programming is Fun

- The counter value changes from 1 to 3 as the range starts at 1 and ends at 3

For Loop

- **Lists are one of the built-in data types in Python used to store collections of data**
- **list(range(4)) in the command line will produce the following list**

  [0, 1, 2, 3]
- **list(range(1, 5)) will produce the list**

  [1, 2, 3, 4]
- **list(range(1, 10, 2)) will produce the list**

  [1, 3, 5, 7, 9]
- **The values contained in a list can be visited by running a for loop**

For Loop

- **The syntax of for loop for processing a list is**

for <variable> in <sequence>:

<do something with variable>

eg:- for counter in list(range(1,10)):

print(str(counter), end =" ")

- **Will produce the output**

1 2 3 4 5 6 7 8 9

eg:- for counter in list(range(1,11,3)):

print(str(counter), end =" ") **produces**

1 4 7 10

For Loop

eg:- for character in "Hi there!":

print(character)

Will produce the output

H
i

t
h
e
r
e
!

For Loop

- **In certain cases it is required to count down from an upper bound to a lower bound**

```
for counter in range(10, 0, -1):
    print(str(counter), end = " ")
```

- **Will produce the output**

10 9 8 7 6 5 4 3 2 1

The While Loop

- **In many situations the number of iterations in a loop is unpredictable**
- **The loop completes its execution only when a condition changes and this type of iteration is called conditional iteration**
- **While loop is a type of conditional iteration**
- **Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue**
- **If the continuation condition is true the statements within the loop are executed**
- **If the continuation condition is false the loop ends**

The While Loop

- **The Syntax of while loop is**

  while <condition>:

  <sequence of statements>

- **Something eventually has to happen within the body of the loop to make the loop's continuation condition become false**
- **Otherwise the loop will continue forever making it an infinite loop**

The Break Statement

- **The break statement in Python is used to exit out of a loop either a for or while loop prematurely before the loop has iterated through all its items or reached its condition**
- **When the break statement is executed the program immediately exits the loop and the control moves to the next line of code after the loop**
- **If we use break statement in the innermost loop in nested loops then the control of the program is terminated only from the innermost loop**

Nested Loops

- **We can use one loop inside another loop and this is called nested loop**
- **The numbers of iterations in nested loop will be equal to the number of iterations in the outer loop multiplied by the number of iterations in the inner loop**
- **The nested loop can be two for loops or two while loops or a while loop inside a for loop or for loop inside a while loop**
- **Nested loops are useful in multiplications, array and matrix processing**

Nested Loops

- **Eg: creating a multiplication table**

```
for i in range(1,11):
    for j in range(1,11):
        print(i, "X", j ,"=", i*j)
```

- **Will give the output**

1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7

Formatting Text for Output

- "%6s" % "four“ in the command line will produce the output

  ' four’ and

  "%-6s" % "four“ will produce 'four '
- The first line of code right-justifies the string by padding it with two spaces to its left
- The second line of code left-justifies by placing two spaces to the string’s right
- The syntax of formatting is

  <format string> % <datum>

Formatting Text for Output

- **The format string can contain string data and other information about the format of the datum**
- **The notation %<field width>s in the format string**
- **When the field width is positive the datum is right-justified and 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**
- **To format integers and decimal numbers the letters d and f are used instead of s**

Formatting Text for Output

- To format a sequence of data values a format string is constructed that includes a format code for each datum and place the data values in a tuple following the % operator
- The syntax of the format string is

  <format string> % (<datum-1>, …, <datum-n>)

eg:- print(“%-3d%12d” % (exponent, 10 ** exponent))

- The first column is left-justified in a field width of 3 and the second column is right-justified in a field width of 12
- The format information for data type float has the form

  %<field width>.<precision>f

Formatting Text for Output

eg: - “%6.3f” % 3.14 gives the output ' 3.140‘

- **In the command line it says to use a field width of 6 and a precision of 3 to format the float value 3.14**
- **Python adds a digit of precision to the number’s 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**

Logical Operators

- **Python has three Boolean or logical operators *and* *or* and *not***
- **Both the *and* operator and the *or* operator needs two operands**
- **The truth table for and, or and not operators are shown here**

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

-

Logical Operators

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

>>> A = True

>>> B = False

>>> A and B

False

>>> A or B

True

>>> not A

False

Random Numbers

- **Random numbers can be generated in Python using the function randit()**
- **The function randint returns a random number from among the numbers between the two arguments supplied and including those numbers**
- **To use randit()function the random package should be imported to the program**

```
import random
print(random.randint(1, 10))
```

```
>>> 7
```

Random Numbers

- **The program will produce a number between 1 and 10 including 1 and 10**
- **The following code will generate 8 random numbers between 1 and 10 including 1 and 10**

```
import random
for roll in range(8):
    print(random.randint(1, 10), end=" ")
```

```
>>> 3  6  4  10  9  8  6  4
```

Module 2

Strings

W	e	l	c	o	m	e		t	o		t	h	e		c	l	a	s	s
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

- **A string is a sequence of zero or more characters**
- **With strings the length of the string and the positions of the individual characters within the string are important**
- **Python's len function returns the number of characters in the string**

```
>>> len("Welcome to the class")
20
```

Strings

- **The subscript operator is used to access each character in a string**

```
>>> noun = "Welcome to the class"
>>> noun[1]
        'e'
```

- **The integer expression also called index in brackets indicates the position of the particular character in the string**

```
>>> noun[len(noun)-1]
        's'
```

Strings

- **The index range from 0 to the length minus 1**
- **Python allows negative subscript values to access characters at or near the end of a string**
- **The program counts backward from -1 to access characters from the right end of the string**

```
>>> noun[-1]
        's'
>>> noun[-3]
        'a'
```

Subtrings

- **Python's subscript operator can be used to obtain a substring of a string**
- **The process of obtaining a substring from a string is called slicing**
- **To extract a substring a colon (:) is used and an integer value can be used on either sides of the colon**

Subtrings

```
>>> noun = "myfile.txt"
>>> noun[0:]
        'myfile.txt'
>>> noun[0:1]
        'm'
>>> noun[0:2]
        'my'
>>> noun[:len(noun)]
        'myfile.txt'
>>> noun[-3:]
        'txt'
```

Subtrings

```
fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]

for fileName in fileList:

    if ".txt" in fileName:

        print(fileName)
```

- Will produce the output

  myfile.txt

  yourfile.txt

- The in operator in python when used the left operand is a target substring and the right operand is the string to be searched

Subtrings

- **The in operator returns True if the target string is somewhere in the search string or False otherwise**
- **The code segment traverses a list of filenames and prints just the filenames that have a .txt extension**

Data Encryption

- **Many applications now use data encryption to protect information that is transmitted**
- **FTP and HTTP has encrypted versions FTPS and HTTPS which are secure versions for file transfer and Web page transfer**
- **The sender encrypts a message by translating it to a secret code called a cipher text**
- **At the other end the receiver decrypts the cipher text back to its original plaintext form**
- **Both sender and receiver must have at their disposal one or more keys that allow them to encrypt and decrypt messages**

Data Encryption

- **This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence**
- **For positive distances the method wraps around to the beginning of the sequence to locate the replacement characters for those characters near its end**
- **For example if the distance value of a encryption equals five characters the string “apple” would be encrypted as “fuuqj”**

Data Encryption

- **To decrypt this text back to plaintext we apply a method that uses the same distance value but looks to the left of each character for its replacement**
- **This decryption method wraps around to the end of the sequence to find a replacement character for one near its beginning**
- **Python scripts implement the encryption and decryption method for any strings that contain lowercase letters and for any distance values between 0 and 26**

Data Encryption

- **The ord function returns the ordinal position of a character value in the ASCII sequence whereas chr returns the character value for the ordinal position**

Number System

- **The decimal number system use the base as ten**
- **The binary number system use the base as two**
- **The binary number is used to represent all information in a digital computer**
- **The binary number is represented using 0 and 1**

247_{10} in Decimal is equal to 1111 0111 $_2$ in Binary

$247 = 2 * 10^2 + 4 * 10^1 + 7 * 10^0$

$2 * 100 + 4 * 10 + 7 * 1$

$200 + 40 + 7 = 247$

Converting Binary to Decimal

- **Each digit or bit in a binary number has a positional value that is a power of 2**
- **1111 0111 =**
- **$1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$**
- **1 * 128 + 1*64 + 1 * 32 + 1* 16 +0 * 8 + 1*4 + 1*2 + 1*1**
- **128 + 64 + 32 + 16+ 0 + 4+ 2 + 1 = 247**
- **In computing the value of a binary number the values of the positions occupied by 0s are ignored and add the positional values of the positions occupied by 1s**

Converting Binary to Decimal

```
bstring = input("Enter a string of bits: ")

decimal = 0

exponent = len(bstring) - 1

for digit in bstring:

    decimal = decimal + int(digit) * 2 ** exponent

    exponent = exponent - 1

print("The integer value is", decimal)
```

Converting Binary to Decimal

- **The program uses a loop that accumulates the sum of a set of integers**
- **The exponent that corresponds to the position of the string's leftmost bit is the length of the bit string minus 1**
- **The loop visits the digits in the string from the first to the last (left to right) also counting from the largest exponent of 2 down to 0 as it goes**
- **Each digit is converted to its integer value (1 or 0) multiplied by its positional value and the result is added to the ongoing total.**
- **A positional value is computed by using the ** operator**

Converting Decimal to Binary

```
decimal = int(input("Enter a decimal integer: "))
if decimal == 0:
    print (0)
else:
    bstring = ""
    while decimal > 0:
        remainder = decimal % 2
        decimal = decimal // 2
        bstring = str(remainder) + bstring
    print("The binary representation is", bstring)
```

Converting Decimal to Binary

- **Decimal to Binary conversion uses Division method**
- **This Program repeatedly divides the decimal number by 2**
- **After each division the remainder (either a 0 or a 1) is placed at the beginning of a string of bits**
- **The quotient becomes the next dividend in the process**
- **The string of bits is initially empty and the process continues while the decimal number is greater than 0**

Strings

- **A method is always called with a given data value called an object which is placed before the method name in the call**
- **The syntax of a method call is**

  <an object>.<method name>(<argument-1>, …, <argument-n>)
- **Methods can expect arguments and return values**
- **A method knows about the internal state of the object with which it is called**
- **In Python all data values are in fact objects and every data type includes a set of methods to use with objects of that type**
- **The following are some of the useful string methods**

Strings

s = "Today is Monday"

len(s)

15

s.center(20)

' Today is Monday ‘

s.count('o')

2

s.endswith("day")

True

Strings

s.startswith("To")

True

s.find('is')

6

s.isalpha()

False

'Today'.isalpha()

True

Strings

```
"934".isdigit()

True

words = s.split()

words

['Today', 'is', 'Monday']

"".join(words)

'TodayisMonday'

" ".join(words)
```

Strings

'Today is Monday‘

s.lower()

'today is monday'

s.upper()

'TODAY IS MONDAY'

s.replace('o', 'i')

'Tiday is Minday'

" Today is Monday ".strip()

'Today is Monday'

Strings

- **The complete list of the string methods can be seen by entering dir(str) at the shell prompt**
- **The documentation about the string methods can be accessed by entering help(str) at the shell prompt**
- **The split method can be used to extract a filename's extension**

```
>>>"myfile.txt".split(".")
['myfile', 'txt']
```

Text Files

- **A text file is a software object that stores data on a permanent medium such as a disk, CD, or flash memory**
- **The main advantages of taking input data from a file compared to keyboard is that**

1. **The data set can be much larger**
2. **The data can be input much more quickly and with less chance of error**
3. **The data can be used repeatedly with the same program or with different programs**

Text Files

- **Python programs can read data from an input file and also output data to a text file**
- **All data output to or input from a text file must be strings**
- **Numbers must be converted to strings before output and these strings must be converted back to numbers after input**
- **Data can be output to a text file using a file object**

Text Files

- **Python’s open function opens a file in different mode**

  f = open(“myfile.txt”, 'w')

- **Here the file called myfile.txt is opened in write mode**
- **If the file does not exist a new file is created with the given name**
- **If the file already exists Python opens it**
- **String data are written to a file using the method write with the file object**

  f.write(“First line.\nSecond line.\n”)

- **The file object writes two sentences to the file**
- **At last the file is closed using f.close()**

Writing numbers to a File

```
f = open("numbers2.txt", 'w')

for count in range(500):

    f.write(str(count) + "\n")

f.close()
```

- The program opens a file called numbers2.txt and writes numbers upto 499 into the file and closes the file
- The program writes one number in a line
- The numbers are integers so they are converted to string before writing to the file using str()
- The strings are written to a file with a newline character

Reading numbers from a File

- File stores the data as strings
- The programmer must convert them to the appropriate types before manipulating them

```
f = open("numbers.txt", 'r')
sum = 0
for line in f:
    line = line.strip()
    number = int(line)
    sum += number
print("The sum is", sum)
```

Reading numbers from a File

- **The data is read with a for loop**
- **The loop accesses a line of text on each pass**
- **While converting the line to integer the string method strip is used to remove the newline character**
- **The int function converts string to the integer value**
- **In case the numbers in a text file are separated by spaces the string method split is used to obtain a list of the strings representing these integers**
- **Then process each string in this list with another for loop**

Reading numbers from a File

```
f = open("numbers.txt", 'r')

sum = 0

for line in f:

    wordlist = line.split()

    for word in wordlist:

        number = int(word)

        print(word)

        sum += number

print("The sum is", sum)
```

Reading text from a File

```
f = open("myfile.txt", 'w')

s = input("Enter the sentence ")

words = s.split(" ")

for word in words:

    f.write(word + "\n")

    print(word)

f.close()
```

- The program accepts a long sentence and split the sentence into words and each word is written into a file line by line so that one line has one word

Reading text from a File

```
f = open("myfile.txt", "r")

for line in f:

    print(line.strip())  # .strip() to remove newline characters

f.close()
```

- The program opens the file and reads the words one by one from each line and displays in the screen
- The strip function is used to remove the the newline character

Accessing and Manuplating Files and Directories on Disk

```
import os

currentDirectoryPath = os.getcwd()

listOfFileNames = os.listdir(currentDirectoryPath)

print(currentDirectoryPath)

for name in listOfFileNames:

    if ".py" in name:

        print(name)
```

- The current working directory is got from OS
- All python programs in the current directory is printed out

Lists

- **A list is a sequence of data values called items or elements**

The following are some real world examples of lists

- **A shopping list for the grocery store**
- **A guest list for a wedding**
- **A recipe, which is a list of instructions**
- **A text document, which is a list of lines**
- **The words in a dictionary**
- **The names in a phone book**
- **Each of the items in a list is ordered by position**

Lists

- **Each item in a list has a unique index that specifies its position**
- **The index of the first item is 0 and the index of the last item is the length of the list minus 1**
- **In Python a list is written as a sequence of data values separated by commas**
- **The entire sequence is enclosed in square brackets ([and])**

  [2000, 2004, 2008, 2012]

  ['apples', 'oranges', 'cherries','grapes']

  [ ]

Lists

- **Each item in a list has a unique index that specifies its position**
- **The index of the first item is 0 and the index of the last item is the length of the list minus 1**
- **In Python a list is written as a sequence of data values separated by commas**
- **The entire sequence is enclosed in square brackets ([and])**

  [2000, 2004, 2008, 2012]

  ['apples', 'oranges', 'cherries','grapes']

  [ ]

Lists

- **We can also use other lists as elements in a list thereby creating a list of lists**

  [[5, 9], [541, 78], [456, 763]]

- **When the Python interpreter evaluates a list literal each of the elements is evaluated first**
- **When the element is a variable or any other expression its value is included in the list**

```
import math
x = 2
[x, math.sqrt(x)]
[2, 1.414213562373095]
```

Lists

- **List of integers can be created using the range and list functions**

```
>>> first = [1, 2, 3, 4]
>>> second = list(range(1, 5))
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
```

- **The function len can be used to find the length of a list**

```
>>> len(second)
4
```

Lists

- **The subscript operator [] can be used with list to access elements at specific positions**

```
>>> first[0]
1
>>> first[2 : 4]
[3, 4]
```

- **Concatenation (+) can be done on lists**

```
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
```

- **Equality (==) can be performed on lists**

Lists

```
>>> first == second
```

```
True
```

- To print the contents of a list without the brackets and commas a for loop can be used

```
for element in [1, 2, 3, 4]:
    print(element, end=" ")
```

```
1 2 3 4
```

- The in operator can be used to detect the presence or absence of a given element in a list

Lists

>>> 4 in [1, 2, 3, 4]

True

>>> 5 in [1, 2, 3, 4]

False

Lists

- **A list is changeable that is it is mutable**
- **At any point in its lifetime elements can be inserted, removed, or replaced**
- **The list itself maintains its identity, but its state its length and its contents can change**
- **The subscript operator is used to replace an element at a given position**

```
>>> first = [1, 2, 3, 4]
>>> first
[1, 2, 3, 4]
```

Lists

```
>>> first[2] = 0
>>> first
[1, 2, 0, 4]
```

- This program replace each number in a list with its square

```
numbers = [2, 3, 4, 5]
index = 0
while index < len(numbers):
    numbers[index] = numbers[index] ** 2
    index += 1
print(numbers)
```

Lists

[4, 9, 16, 25]

Lists

```
sentence = "This example has five words."

words = sentence.split()

index = 0

while index < len(words):

    words[index] = words[index].upper()

    index += 1

print(words)
```

['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']

- The string method split is used to extract a list of the words in a sentence

Lists

- **These words are then converted to uppercase letters within the list**

```
numbers = list(range(6))

print(numbers)

numbers[0:3] = [11, 12, 13]

print(numbers)

>>>[0, 1, 2, 3, 4, 5]

>>>[11, 12, 13, 3, 4, 5]
```

- **The program replaces the first three elements of a list with new elements**

List methods for Inserting and Removing Elements

- **The method insert expects an integer index and the new element as arguments**
- **When the index is less than the length of the list the new element is placed before the existing element at that index after shifting elements to the right by one position**
- **When the index is greater than or equal to the length of the list the new element is added to the end of the list**

```
>>> first = [1, 2, 3, 4]
>>> first.insert(2, 15)
```

List methods for Inserting and Removing Elements

```
>>> first

[1, 2, 15, 3, 4]

>>> first.insert(5, 25)

>>> first

[1, 2, 15, 3, 4, 25]
```

- The method append adds an element to the end of the list
- Whereas extend function adds multiple elements to the list

```
>>> first = [1, 2, 3, 4]

>>> first.append(5)
```

List methods for Inserting and Removing Elements

```
>>> first
[1, 2, 3, 4, 5]
>>> first.extend([6, 7, 8])
>>> first
[1, 2, 3, 4, 5, 6, 7, 8]
```

- The method pop is used to remove an element from a list
- If the position is not specified pop removes and returns the last element
- If the position is specified pop removes the element at that position and returns it

List methods for Inserting and Removing Elements

```
>>> first
[1, 2, 3, 4, 5]
>>> first.extend([6, 7, 8])
>>> first
[1, 2, 3, 4, 5, 6, 7, 8]
```

- The method pop is used to remove an element from a list
- If the position is not specified pop removes and returns the last element
- If the position is specified pop removes the element at that position and returns it

List methods for Inserting and Removing Elements

```
>>>example
   [1, 2, 3, 4, 5, 6]
>>> example.pop()
6
>>> example
[1, 2, 3, 4, 5]
>>> example.pop(0)
1
>>> example
[2, 3, 4, 5]
```

Searching a List

- **The in operator determines an element's presence in a list**
- **The method index is used to locate an element's position in a list**
- **In the program the in operator is used to test for the presence of an element and then the index method returns the position of the element in the List**

```
first = [10, 15, 20, 25, 30, 35, 40, 45, 50]

target = 45

if target in first:

    print(first.index(target))

else:
```

Searching a List

```
    print(-1)
```

```
>>> 7
```

- **A list can be numerically or alphabetically arranged using the sort function**

```
s = [20, 98, 12, 65, 24, 76]
s.sort()
print(s)
```

```
>>> [12, 20, 24, 65, 76, 98]
```

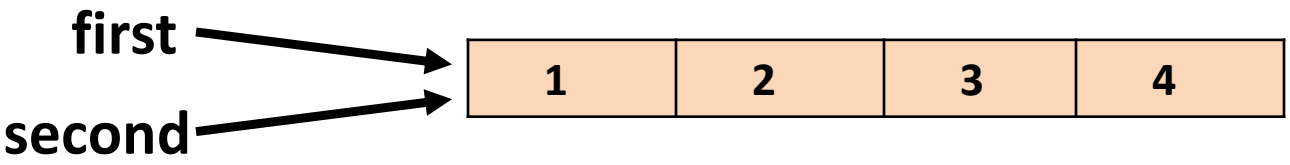
- **The elements in the list are numerically arranged in the ascending order**

Sorting a List

```
f = ["banana", "kiwi", "apple", "fig"]

f.sort()

print(f)

>>>['apple', 'banana', 'fig', 'kiwi']

f.sort(key=len)

print(f)

>>>['fig', 'kiwi', 'apple', 'banana']

f.sort(reverse=True)

print(f)

>>>['kiwi', 'fig', 'banana', 'apple']
```

Aliasing

```
>>> first = [1, 2, 3, 4]
>>> second = first
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
>>> first[2] = 50
>>> first
[1, 2, 50, 4]
>>> second
[1, 2, 50, 4]
```

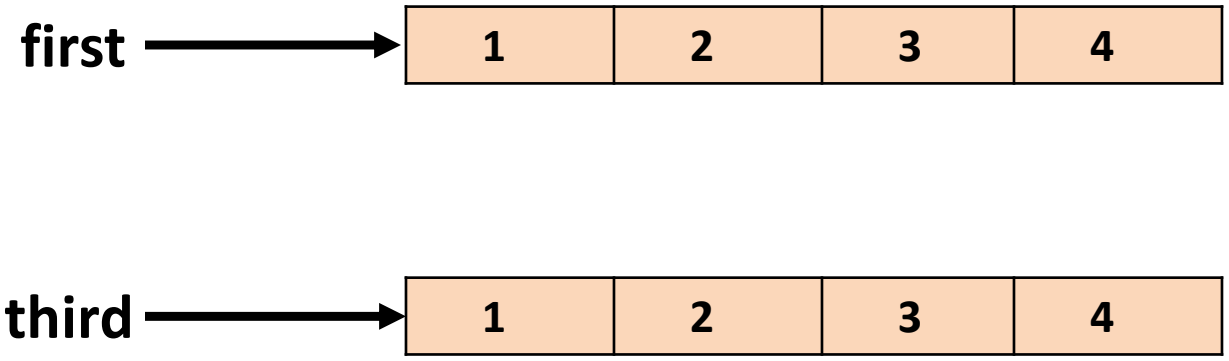


Aliasing

- **In this example, a single list object is created and modified using the subscript operator**
- **When the third element of the list named first is replaced the third element of the list named second is replaced also**
- **The assignment second = first the variables first and second refer to the exact same list object**
- **This phenomenon is known as aliasing**
- **In the next session we create a new object(third) and copy the contents of the original(first) to it**

Aliasing

- **The variables first and third refer to two different list objects, although their contents are initially the same**



Aliasing

```
>>> first = [1, 2, 3, 4]
>>> first
[1, 2, 3, 4]
>>> third = []
>>> for element in first:
        third.append(element)
>>> third
[1, 2, 3, 4]
>>> first[2] = 100
>>> first
[1, 2, 100, 4]
>>> third
[1, 2, 3, 4]
```

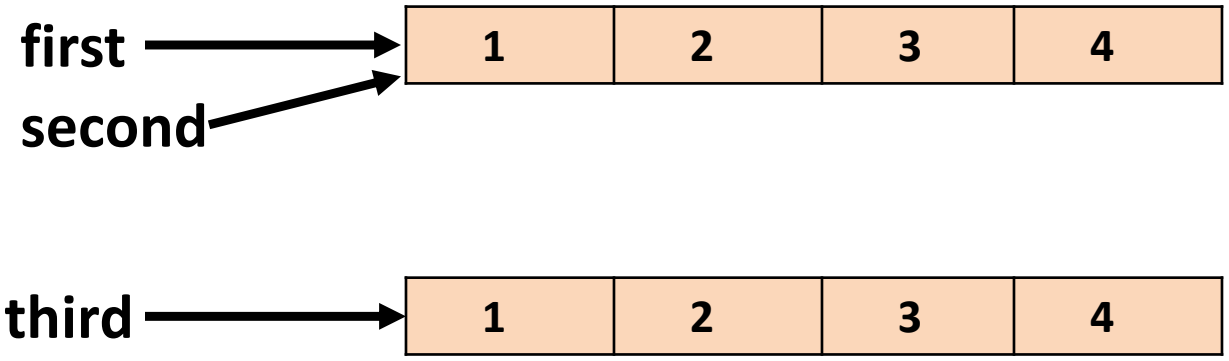
Equality

- **The == operator returns True if the variables are aliases for the same object**
- **The equality operator == also returns True if the contents of two different objects are the same**
- **The first relation is called object identity whereas the second relation is called structural equivalence**
- **Python's is operator can be used to test for object identity**
- **It returns True if the two operands refer to the exact same object and it returns False if the operands refer to distinct objects even if they are structurally equivalent**

Aliasing

```
>>> first = [1, 2, 3, 4]
>>> second = first
>>> third = [1, 2, 3, 4]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```

Equality



Tuples

- **A tuple is a type of sequence that resembles a list**
- **A tuple is immutable so it cannot be updated and elements cannot be inserted or deleted**
- **A tuple literal is indicated in Python by enclosing its elements in parentheses instead of square brackets**
- **Many of the operators and functions used with lists can also be used with tuples**
- **A tuple should be used instead of a list where the structure will not change**
- **eg:- vowels**

Tuples

```
>>>fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> tuple(fruits)
('apple', 'banana')
```

Defining Simple Functions

- **The user can define his own function**
- **These functions will have arguments and return value**
- **The next function computes the average value of a list of numbers**

```
def average(list):
    sum = 0
    for number in list:
        sum += number
    return sum / len(list)
>>> average([2, 5, 6, 8])
5.25
```

Defining Simple Functions

- **The number and positions of the arguments of a function call should match the number and positions of the parameters in that function's definition**
- **The syntax of a function definition contains a header and a body**
- **The header consists of the reserved word def followed by the function's name, followed by a parenthesized list of parameters and a colon**

def <function name>(<parameter-1>,…,<parameter-n>):

<body>

Boolean Functions

- **A Boolean function usually tests its argument for the presence or absence of some property**
- **The function returns True if the property is present or False otherwise**
- **The following Boolean function odd tests a number to see whether it is odd or not**

```
def odd(x):
    if x % 2 == 1:
        return True
    else:
        return False
```

Boolean Functions

>>> odd(5)

True

>>> odd(8)

False

- **This function has two possible exit points in either of the alternatives within the if/else statement**

Defining a main function

- **A special function named main can be used as the entry point for the scripts**
- **The main function usually expects no arguments and returns no value**
- **Its sole purpose is to take inputs process them by calling other functions and print the results**
- **The main function prompts the user for a number calls the square function to compute its square and prints the result**
- **We can define the main and the square functions in any order**
- **The main is called as the last step in the script**

Defining a main function

```
def main():
    number = float(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)
def square(x):
    return x * x
main()
```

```
>>> Enter a number: 7
The square of 7.0 is 49.0
```

Dictionaries

- **A dictionary organizes information by association and not by position**
- **Lists organize their elements by position**
- **In Python a dictionary associates a set of keys with data values**
- **A Dictionary comprise a set of words and their definitions**
- **A Python dictionary is written as a sequence of key/value pairs separated by commas**
- **These pairs are called entries**
- **The entire sequence of entries is enclosed in curly braces ({ and })**

Dictionaries

- **A colon (:) separates a key and its value**

  A Phone Book {'Harish':'9784753321', 'Thomas':'9835167743'}

  Personal information: {'Name':'Gopal', 'Age':24}

- **An empty dictionary can be created like this { }**
- **A new key/value pair can be added to a dictionary by using the subscript operator [ ]**
- **The syntax is**

  <dictionary>[<key>] = <value>

- **The following code creates a dictionary and changes its value**

Dictionaries

```
>>>phoneBook = { }

>>> phoneBook["name"] = "Kurian"

>>> phoneBook["number"] = 9654132700

>>> phoneBook

{'name': 'Kurian', 'number': 9654132700}

>>> phoneBook["number"] = 3651487462

>>> phoneBook

{'name': 'Kurian', 'number': 3651487462}
```

- To delete an entry from a dictionary pop method can be used

Dictionaries

>>>phoneBook.pop("number")

3651487462

>>> phoneBook

{'name': 'Kurian'}

- **A for loop can be used with a dictionary to print all the key/value pairs**
- **The entries are represented as tuples within the list**
- **A tuple of variables can access the key and value of each entry in this list within a for loop**

Dictionaries

```
phoneBook = {"Thomas": 9876547823, "Shibu": 3456732184,
"Aby": 5478923154, "Cibu": 2897534729}
for (key, value) in phoneBook.items():
    print(key, value)
```

- The list can be sorted based on the key and can be displayed

```
theKeys = list(phoneBook.keys())
theKeys.sort()
for key in theKeys:
    print(key, phoneBook[key])
```

Hexadecimal System

- **The program visits each digit in the hexadecimal number selects the corresponding four bits that represent that digit in binary and adds these bits to a result string**
- **We maintain the set of associations between hexadecimal digits and binary digits in a dictionary then we can just look up each hexadecimal digit's binary equivalent and fetch the binary equivalent**
- **Such a dictionary is called a lookup table**

Hexadecimal System

```
hexa = input("Enter the hexa number: ")
hexToBinaryTable = {'0':'0000', '1':'0001', '2':'0010','3':'0011',
                    '4':'0100', '5':'0101', '6':'0110', '7':'0111', '8':'1000',
                   '9':'1001', 'A':'1010', 'B':'1011', 'C':'1100', 'D':'1101',
                    'E':'1110', 'F':'1111'}
def convert(number):
   binary = ''
   for digit in number:
      binary = binary + hexToBinaryTable[digit]
   print(binary)
convert(hexa)
```

Mode of a list of values

- **The mode of a list of values is the value that occurs most frequently**
- **In the program the input is a list of words from a text file and prints their mode**
- **The script uses a list and a dictionary**
- **The list is used to obtain the words from the file**
- **The dictionary associates each unique word with the number of its occurrences in the list**
- **The script also uses the function max to compute the maximum of values**

Mode of a list of values

- Max function returns the largest value contained in the list

```
fileName = input("Enter the filename: ")

f = open(fileName, 'r')

words = []

for line in f:

    wordsInLine = line.split()

    for word in wordsInLine:

        words.append(word.upper())

theDictionary = {}
```

Mode of a list of values

```
for word in words:
    number = theDictionary.get(word, None)
    if number == None:
        theDictionary[word] = 1
    else:
        theDictionary[word] = number + 1
        print(word,theDictionary[word])
theMaximum = max(theDictionary.values())
```

Mode of a list of values

```
for key in theDictionary:
    if theDictionary[key] == theMaximum:
        print("The mode is", key)
        break
```

Design with functions

- **Abstraction is the process of hiding all relevant data about an object in order to reduce complexity and increase efficiency**
- **An abstraction hides detail and thus allows a person to view many things as just one thing**
- **Making coffee with a coffee machine is a good example of abstraction**
- **We need to know how to use your coffee machine to make coffee**
- **We need to provide water and coffee beans switch it on and select the kind of coffee we want to get**

Design with functions

- **We don’t need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee**
- **We don’t need to know the ideal temperature of the water or the amount of ground coffee we need to use**
- **Someone else worried about that and created a coffee machine that now acts as an abstraction and hides all these details**
- **We just interact with a simple interface that doesn’t require any knowledge about the internal implementation**
- **The fuctions in Python language provide an abstraction that hides the internal implementation details**

Design with functions

1. **Functions Eliminate Redundancy**
2. **Functions Hide Complexity**
3. **Functions Support General Methods with Systematic Variations**
4. **Functions Support the Division of Labor**

Functions Eliminate Redundancy

- **Functions serve as abstraction mechanisms by eliminating redundant or repetitious code**
- **To make the concept more clear let's look at a function named sum which returns the sum of the numbers within a given range of numbers**

```
def sum(lower, upper):
  result = 0
  while lower <= upper:
    result += lower
    lower += 1
  return result
```

Functions Eliminate Redundancy

- **If the sum function does not exist the programmer would have to write the entire program every time a summation is required**
- **In a program that must calculate multiple summations the same code would appear multiple times which will cause redundancy**
- **Redundancy will cause many problems as the programmer decides to improve the algorithm by adding a new feature he has to revise each instance of the redundant code throughout the entire program which will be a maintenance nightmare**

Functions Eliminate Redundancy

- **By relying on a single function definition instead of multiple instances of redundant code the programmer has to write only a single program in just one place**
- **When the programmer needs to debug, repair or improve the function he needs to edit and test only the single function definition**

Functions Hide Complexity

- **Functions serve as abstraction mechanisms by hiding complicated details**
- **The idea of summing a range of numbers is simple but the code for computing a summation is complex**
- **It is not only about the amount or length of the code but also about the number of interacting components**
- **There are three variables to manipulate, as well as count-controlled loop logic to construct**

Functions Support General Methods with Systematic Variations

- **An algorithm is a general method for solving a class of problems**
- **The individual problems that make up a class of problems are known as problem instances**
- **The problem instances for the summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed**
- **When an algorithm is designed it should be general enough to provide a solution to many problem instances not just one or a few of them**

Functions Support General Methods with Systematic Variations

- **A function should provide a general method with systematic variations**
- **The problem instances are the data sent as arguments to the function**
- **Additional arguments can broaden the range of problems that are solvable**
- **The sum function could take a third argument that specifies the step to take between numbers in the range**

Functions Support the Division of Labor

- **In a computer program functions can enforce a division of labor**
- **Each function performs a single coherent task such as computing a summation or formatting a table of data for output**
- **Each function is responsible for using certain data computing certain results, and returning these to the parts of the program that requested them**
- **Each of the tasks required by a system can be assigned to a function including the tasks of managing or coordinating the use of other functions**

Problem Solving with Top-Down Design

- **The Top-down design strategy starts with a global view of the entire problem and break the problem into smaller more manageable subproblems**
- **This process is a problem decomposition**
- **Problem decomposition may continue down to lower levels because a subproblem might in turn contain two or more lower-level problems to solve**
- **As functions are developed to solve each subproblem the solution to the overall problem is gradually filled out in detail**
- **This process is also called stepwise refinement**

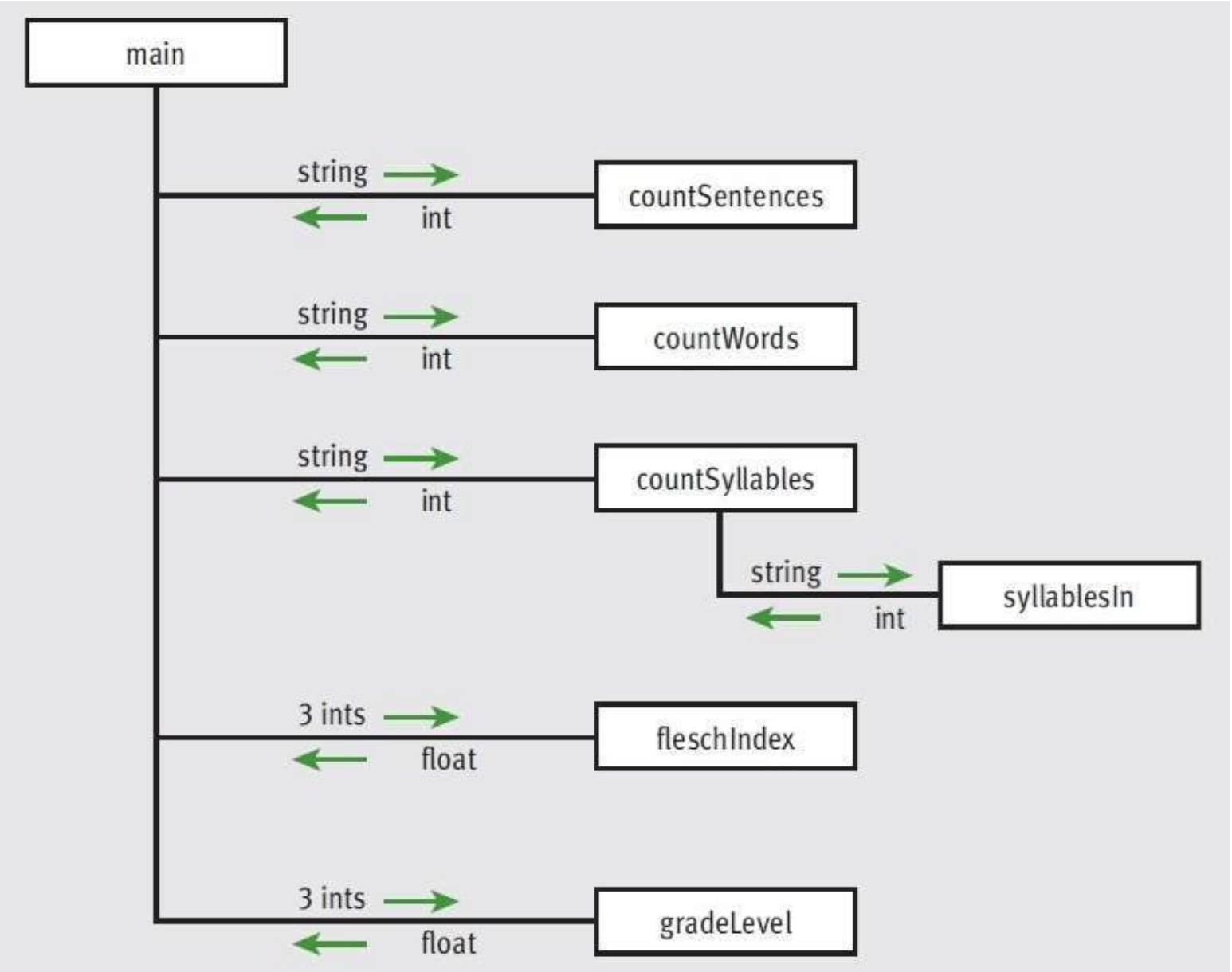
Problem Solving with Top-Down Design

- **The programs we have seen were simple that they could be decomposed into three parts the input of data its processing and the output of results**
- **These parts require only one or two statements of code and they all appeared in a single sequence of statements**
- **When problems became more complicated decomposition and assignment to additional programmer defined functions are required**

The Design of the Text-Analysis Program

- **In the Text Analysis Program the processing of input is complex so that to decompose into smaller subprocesses, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores**
- **New functions are developed for each of these computational tasks**
- **A structure chart is a diagram that shows the relationships among a program's functions and the passage of data between them**

The Design of the Text-Analysis Program



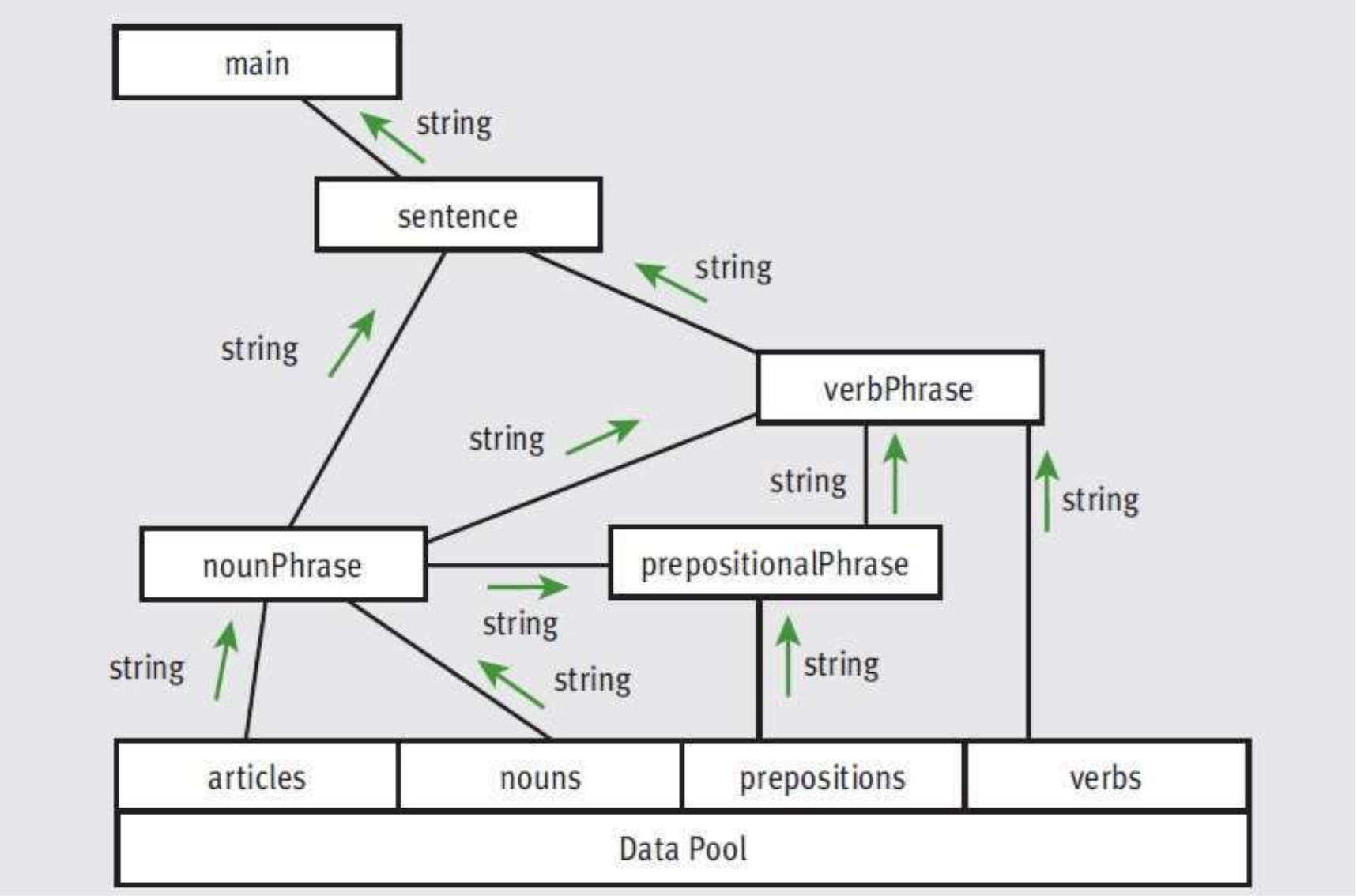
The Design of the Text-Analysis Program

- **Each box in the structure chart is labeled with a function name**
- **The main function at the top is where the design begins and decomposition leads us to the lower-level functions on which main depends**
- **The lines connecting the boxes are labeled with data type names and arrows indicate the flow of data between them**
- **The function countWords takes a string as an argument and returns the number of words in that string**

The Design of the Sentence-Generator Program

- **The sentence-generator program consists of a main loop in which sentences are generated a user-specified number of times**
- **The problem is decomposed by simply following the grammar rules for phrases.**
- **A sentence is generated by generating a noun phrase followed by a verb phrase and so on**
- **Each of the grammar rules poses a problem that is solved by a single function**
- **The top-down design flows out of the top-down structure of the grammar**

The Design of the Sentence-Generator Program



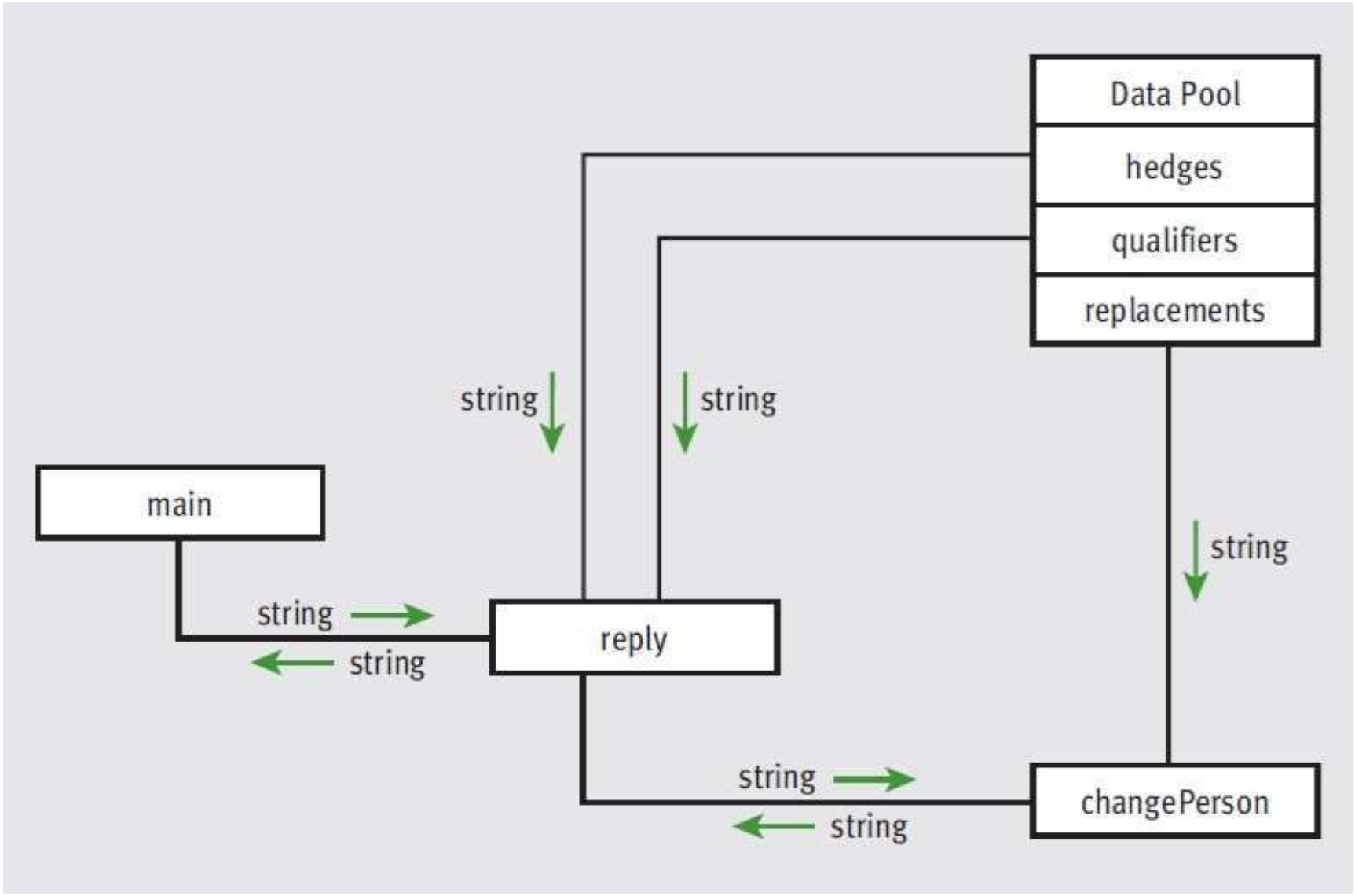
The Design of the Sentence-Generator Program

- **In the sentence generator the structure of the problem comes from the grammar rules**
- **The functions in the text analyzer all receive data from the main function via parameters or arguments**
- **The functions in the sentence generator receive their data from a common pool of data**
- **Using a common pool of data rather than function arguments in this case simplifies the design and makes program maintenance easier**

The Design of the Doctor Program

- **The programs main loop takes a user input and print a result**
- **The doctor program actually processes the input by responding to it as an agent would in a conversation**
- **The responsibility for responding is delegated to the reply function**
- **The two functions main and reply have distinct responsibilities**
- **The job of main is to handle user interaction with the program whereas reply is responsible for implementing the “doctor logic” of generating an appropriate reply**

The Design of the Doctor Program



The Design of the Doctor Program

- **The assignment of roles and responsibilities to different actors in a program is also called responsibility-driven design**
- **If there were only one way to reply to the user the problem of how to reply would not be further decomposed**
- **Here there are at least two options reply is given the task of implementing the logic of choosing one of them and asks for help from other functions such as changePerson to carry out each option**
- **Separating the logic of choosing a task from the process of carrying out a task makes the program more maintainable**

The Design of the Doctor Program

- **The data flow scheme used in the doctor program combines the strategies used in the text analyzer and the sentence generator**
- **The doctor program's functions receive their data from two sources**
- **The patient's input string is passed as an argument to reply and changePerson whereas the qualifiers, hedges, and**
- **pronoun replacements are looked up in a common pool of data defined at the beginning of the module**
- **The use of a common pool of data allows the program to grow easily**

Design with Recursive Functions

- **A recursive function is a function that calls itself for execution**
- **To prevent a function from repeating itself indefinitely it must contain at least one selection statement**
- **This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step**
- **Let us see how an iterative function is converted to a recursive function**
- **The function displayRange prints the numbers from a lower bound to an upper bound**

Design with Recursive Functions

```
def displayRange(lower, upper):
    while lower <= upper:
        print(lower)
        lower = lower + 1
```

- The loop's body continues execution while lower <= upper
- When the function executes lower is incremented by 1 but upper never changes
- The loop is replaced with a selection statement, and the assignment statement is replaced with a recursive call of the function

Design with Recursive Functions

```
def displayRange(lower, upper):
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```

- The data value is used to test for the base case that ends the recursive process and also is modified in some way before each recursive step
- In the case of displayRange the value of the argument lower is incremented before each recursive call so that it eventually exceeds the value of the argument upper

Design with Recursive Functions

```
def sum(lower, upper):
    if lower > upper:
        return 0
    else:
        return lower + sum(lower + 1, upper)
```

- The sum function computes and returns the sum of the numbers between these two values
- In the recursive function sum returns 0 if lower exceeds upper (the base case).
- Otherwise the function adds lower to the sum of lower + 1 and upper and returns this result

Design with Recursive Functions

- **To get a better understanding of how recursion works it is helpful to trace a recursion call**
- **We will have a trace for the recursive version of sum function**
- **We add an argument for a margin of indentation and print statements to trace the two arguments and the value returned on each call**
- **The first statement on each call computes the indentation which is then used in printing the two arguments**
- **The value computed is also printed with this indentation just before each call returns**

Design with Recursive Functions

```
def sum(lower, upper, margin):
    blanks = " " * margin
    print(blanks, lower, upper)
    if lower > upper:
        print(blanks, 0)
        return 0
    else:
        result = lower + sum(lower + 1, upper, margin + 4)
        print(blanks, result)
        return result


sum(1, 5, 0)
```

Output

```
1 5
    2 5
        3 5
            4 5
                5 5
                    6 5
                0
            5
        9
    12
  14
15
```

Design with Recursive Functions

- **The displayed pairs of arguments are indented further to the right as the calls of sum proceed**
- **The value of lower increases by 1 on each call whereas the value of upper stays the same**
- **The final call of sum returns 0 and as the recursion unwinds each value returned is aligned with the arguments above it and increases by the current value of lower**

Using Recursive Definitions to Construct Recursive Functions

- **Recursive functions are frequently used to design algorithms for computing values that have a recursive definition**
- **A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases**
- **The Fibonacci sequence is a series of values with a recursive definition**
- **The first and second numbers in the Fibonacci sequence are 1**
- **Each number in the sequence is the sum of its two predecessors**

Using Recursive Definitions to Construct Recursive Functions

- **The Fibonacci series is**

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

- **The recursive definition of the nth Fibonacci number is**

Fib(n) = 1, when n = 1 or n = 2

Fib(n) = Fib(n – 1) + Fib(n – 2), for all n > 2

Using Recursive Definitions to Construct Recursive Functions

```
def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))
terms = int(input("Enter number of terms:"))
for i in range(terms):
    print(recur_fibo(i), end =" ")
```

Recursion in Sentence Structure

- **The structure of sentences in a language can be highly recursive**
- **A noun phrase can be modified by a prepositional phrase which also contains another noun phrase**
- **The nounPhrase function would call the prepositionalPhrase function which in turn calls nounPhrase again and this phenomenon is known as indirect recursion**
- **To keep this process from going on forever nounPhrase must also have the option to not generate a prepositional phrase**
- **The code for a revised nounPhrase function generates a modifying prepositional phrase approximately 25% of the time**

Recursion in Sentence Structure

```
def nounPhrase():
    """Returns a noun phrase, which is an article followed
    by a noun and an optional prepositional phrase."""
    phrase = random.choice(articles) + " " + random.choice(nouns)
    prob = random.randint(1, 4)
    if prob == 1:
        return phrase + " " + prepositionalPhrase()
    else:
        return phrase
```

Infinite Recursion

- **Recursive functions require thorough testing**
- **One design error that might trip up a programmer occurs when the function can continue executing forever a situation known as infinite recursion**
- **Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process**
- **The Python virtual machine eventually runs out of memory resources to manage the process so it halts execution with an error message**

Infinite Recursion

```
def runForever(n):
    if n > 0:
        runForever(n)
    else:
        runForever(n - 1)
runForever(1)
```

- The Python virtual machine keeps calling runForever(1) until there is no memory left
- An infinite recursion in python eventually halts execution with an error message

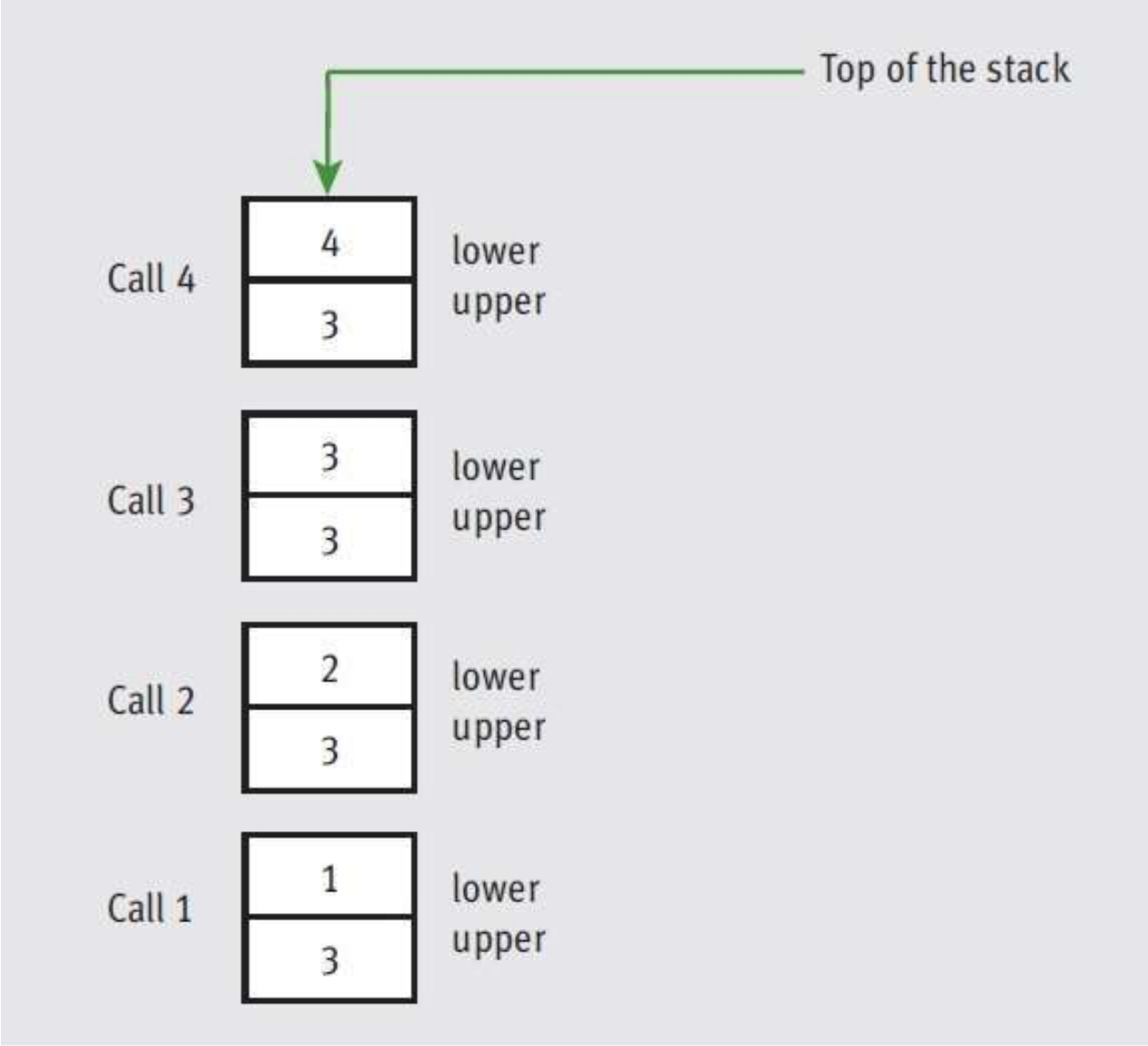
The Costs and Benefits of Recursion

- **In a real computer the Python Virtual Machine(PVM) must devote some overhead to recursive function calls**
- **At program startup the PVM reserves an area of memory named a call stack**
- **For each call of a function the PVM must allocate on the call stack a small chunk of memory called a stack frame**
- **In this type of storage the system places the values of the arguments and the return address for the particular function call**
- **Space for the function call's return value is also reserved in its stack frame**

The Costs and Benefits of Recursion

- When a call returns or completes its execution the return address is used to locate the next instruction in the caller's code and the memory for the stack frame is deallocated
- The stack frames for the process generated by displayRange(1, 3) are shown
- The frames in the figure include storage for the function's arguments only
- When a function invokes hundreds of recursive calls the amount of extra resources required both in processing time and in memory usage can add up to a significant performance hit

The Costs and Benefits of Recursion



The Costs and Benefits of Recursion

- **Due to a design error if the recursion goes infinitely the stack frames are added until the PVM runs out of memory which halts the program with an error message**
- **By contrast, the same problem can often be solved using a loop with a constant amount of memory in the form of two or three variables**
- **Because the amount of memory needed for the loop does not grow with the size of the problem's data set, the amount of processing time for managing this memory does not grow either**

The Costs and Benefits of Recursion

- **It is better to develop recursive solutions when they seem natural particularly when the problems themselves have a recursive structure**
- **Smart compilers also exist that can optimize some recursive functions by translating them to iterative machine code**

Managing a Program's Namespace

- **When we select a variable name that reflect their purpose in a program or the character of the objects in a given problem domain**
- **These variable names are meaningful only to us the human programmers**
- **To the computer, the only meaning of a variable name is the value to which it happens to refer at any given point**
- **A *namespace* is the set of its variables and their values and how can be controlled via good design principles**

Managing a Program's Namespace

- **Let us analyze the namespace of the doctor program**
- **We will focus on the code for the variable replacements and the function changePerson**
- **The names in doctor.py fall into four categories**
- **Module variables**
- **Parameters**
- **Temporary variables**
- **Method names**

Managing a Program's Namespace

- **Module variables:- The names replacements and changePerson are introduced at the level of the module**
- **Although replacements names a dictionary and changePerson names a function they are both considered variables**
- **When module variables are introduced in a program they are immediately given a value**
- **Parameters:- The name sentence is a parameter of the function changePerson**
- **A parameter name behaves like a variable and is introduced in a function or method header**

Managing a Program's Namespace

- **The parameter does not receive a value until the function is called**
- **Temporary variables:- The names words, replyWords, and word are introduced in the body of the function changePerson**
- **Like module variables temporary variables receive their values as soon as they are introduced**
- **Method names:- The names split and join are introduced or defined in the str type**
- **A method reference always uses an object in this case a string followed by a dot and the method name**

Scope

- **The meaning of a word often depends on its surrounding context**
- **For example, in the sports section of the newspaper, the word bat means a stick for hitting baseballs whereas in a story about vampires it means a flying mammal**
- **In a program the context that gives a name a meaning is called its scope**
- **In Python, a name's scope is the area of program text in which the name refers to a given value**
- **Consider the doctor program**

Scope

- **The scope of the temporary variables words, replyWords, and word is the area of code in the body of the function changePerson just below where eachvariable is introduced**
- **The meanings of temporary variables are restricted to the body of the functions in which they are introduced, and are invisible elsewhere in a module**
- **The scope of the parameter sentence is the entire body of the function changePerson**
- **Like temporary variables parameters are invisible outside the function definitions where they are introduced**

Scope

- **The scope of the module variables replacements and changePerson includes the entire module below the point where the variables are introduced**
- **The reply function which calls changePerson might be defined before changePerson in the doctor module**
- **A Python function can reference a module variable for its value it cannot under normal circumstances assign a new value to a module variable**
- **When a Python function reference a module the PVM creates a new temporary variable of the same name within the function**

Scope

```
x = 5
def f():
    x = 10     # Attempt to reset x
    print(x)
f()
print(x)       # No, this displays 5
```

- When the function f is called it does not assign 10 to the module variable x instead it assigns 10 to a temporary variable x
- Once the temporary variable is introduced the module variable is no longer visible within function f
- In any case the module variable's value remains unchanged by the call

Lifetime

- **A computer program has two natures**
- **On the one hand a program is a piece of text containing names that a human being can read for a meaning**
- **Viewed from this perspective variables in a program have a scope that determines their visibility**
- **On the other hand a program describes a process that exists for a period of time on a real computer**
- **Viewed from this other perspective a program's variables have another important property called a lifetime**

Lifetime

- **A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it**
- **When a variable comes into existence storage is allocated for it when it goes out of existence storage is reclaimed by the PVM**
- **Parameters and temporary variables come into existence when they are bound to values during a function call but go out of existence when the function call terminates**
- **The concept of lifetime explains the existence of two variables called x**

Lifetime

- **The module variable x comes into existence before the temporary variable x and survives the call of function f**
- **During the call of f storage exists for both variables so their values remain distinct**

Default Arguments

- **Arguments provide the function's caller with the means of transmitting information to the function**
- **Adding an argument or two to a function can increase its generality by extending the range of situations in which the function can be used**
- **The use of the extra arguments should be optional for the caller of the function**
- **When the function is called without the extra arguments it provides reasonable default values for those arguments that produce the expected results**

Default Arguments

- **Python's range function can be called with one two or three arguments**
- **When all three arguments are supplied they indicate a lower bound an upper bound, and a step value**
- **When only two arguments are given the step value defaults to 1**
- **If a single argument is given the step is assumed to be 1 and the lower bound automatically is 0**
- **The programmer can also specify optional arguments with default values in any function definition**

Default Arguments

- **The syntax of optional arguments is**

  def <function name>(<required args>,

  <key-1> = <val-1>, … <key-n> = <val-n>)

- **The required arguments are listed first in the function header**
- **These are the ones that are essential for the use of the function by any caller**
- **Following the required arguments are one or more default or keyword arguments**
- **When the function is called without these arguments their default values are automatically assigned to them**

Default Arguments

- **When the function is called with these arguments the default values are overridden by the caller's values**
- **When using functions that have default arguments we must provide the required arguments and place them in the same positions as they are in the function definition's header**
- **The default arguments can be supplied**
- **By position where the values are supplied in the order in which the arguments occur in the function header and defaults are used for any arguments that are omitted**

Default Arguments

- **By keyword where one or more values can be supplied in any order using the syntax <key> = <value> in the function call**
- **Here is an example of a function with one required argument and two default arguments**

```
def example(required, option1 = 2, option2 = 3):
    print(required, option1, option2)
>>> example(1)
1   2    3
>>> example(1, 10)
1   10   3
```

Default Arguments

```
>>> example(1, 10, 20)

1   10    20

>>> example(1, option2 = 20)

1    2   20

>>> example(1, option2 = 20, option1 = 10)

1   10   20
```

Higher-Order Functions

- **In Python functions can be treated as data objects**
- **Functions can be assigned to variables passed as arguments to other functions returned as the values of other functions and stored in data structures such as lists and dictionaries**

```
>>> f = abs     # f is an alias for abs
>>>f             # Evaluate f
<built-in function abs>
>>>f(-7)        # Apply f to an argument
7
>>>import math
>>>math.sqrt
    <built-in function sqrt>
```

Higher-Order Functions

```
>>>funcs = [abs, math.sqrt]  # Put the functions in a list
>>>funcs
[<built-in function abs>, <built-in function sqrt>]
>>>funcs[1](2)                    # Apply math.sqrt to 2
1.4142135623730951
```

- Passing a function as an argument to another function is no different from passing any other datum
- The function argument is first evaluated producing the function itself and then the parameter name is bound to this value

Higher-Order Functions

- **The function can then be applied to its own argument with the usual syntax**

```
>>> def example(functionArg, dataArg):
        return functionArg(dataArg)
>>> example(abs, -4)
4
>>> example(math.sqrt, 2)
1.4142135623730951
```

Mapping

- **Mapping process applies a function to each value in a sequence (such as a list, a tuple, or a string) and returns a new sequence of the results**
- **Python includes a map function for this**
- **we have a list named words that contains strings that represent integers**
- **We want to replace each string with the corresponding integer value**
- **The map function easily accomplishes this**

Mapping

```
>>> words = ["231", "20", "-45", "99"]

>>> map(int, words)

<map object at 0x000002F2276ABA90>

>>> words

['231', '20', '-45', '99']

>>> words = list(map(int, words))

>>> words

[231, 20, -45, 99]
```

- The map function builds and returns a new map object which we feed to the list to view the results

Filtering

- A second type of higher-order function is called a filtering
- In this process a function called a predicate is applied to each value in a list
- If the predicate returns True the value passes the test and is added to a filter object
- Otherwise the value is dropped from consideration
- Python includes a filter function

```
def odd(n): return n % 2 == 1
>>> list(filter(odd, range(10)))
[1, 3, 5, 7, 9]
```

Reducing

- **The next example of a higher-order function is called a reducing**
- **We take a list of values and repeatedly apply a function to accumulate a single data value**
- **A summation is a good example of this process**
- **The first value is added to the second value then the sum is added to the third value and so on until the sum of all the values is produced**
- **The Python functools module includes a reduce function that expects a function of two arguments and a list of values**
- **The reduce function returns the result of applying the function**

Reducing

```
>>> from functools import reduce

>>> def add(x, y): return x + y

>>> def multiply(x, y): return x * y

>>> data = [1, 2, 3, 4]

>>> reduce(add, data)

10

>>> reduce(multiply, data)

24
```

Using lambda to Create Anonymous Functions

- **Python includes a mechanism called lambda that allows the programmer to create functions on the fly**
- **A lambda is an anonymous function**
- **It has no name of its own but contains the names of its arguments as well as a single expression**
- **When the lambda is applied to its arguments its expression is evaluated and its value is returned**
- **The syntax of a lambda is**

  lambda <argname-1, ..., argname-n>: <expression>

- **All of the code must appear on one line**

Using lambda to Create Anonymous Functions

- **A lambda cannot include a selection statement because selection statements are not expressions**

```
>>> data = [1, 2, 3, 4, 5]
>>> reduce(lambda x, y: x + y, data)
15
>>> reduce(lambda x, y: x * y, data)
120
```

- **The next example shows the use of range, reduce, and lambda to simplify the definition of the sum function**

Using lambda to Create Anonymous Functions

```
from functools import reduce

def sum(lower, upper):
    if lower > upper:
        return 0
    else:
        return reduce(lambda x, y: x + y, range(lower, upper + 1))

>>> sum(1, 6)
21
```

Module 3

Simple Graphics

- **Python comes with a large array of resources that support graphics operations**
- **A Turtle graphics toolkit provides a simple and enjoyable way to draw pictures in a window and gives an opportunity to run several methods with an object**
- **Turtle graphics were originally developed as part of the children's programming language Logo, created by Seymour Papert and his colleagues at MIT in the late 1960s**
- **Imagine a turtle crawling on a piece of paper with a pen tied to its tail**
- **Commands direct the turtle as it moves across the paper and tell it to lift or lower its tail, turn some number of degrees left or right and move a specified distance**
- **Whenever the tail is down the pen drags along the paper leaving a trail**

Simple Graphics

- **The sheet of paper is a window on a display screen and the turtle is an icon such as an arrowhead**
- **At any given moment in time the turtle is located at a specific position in the window**
- **This position is specified with (x, y) coordinates**
- **The coordinate system for Turtle graphics is the standard Cartesian system with the origin (0, 0) at the center of a window**
- **The turtle's initial position is the origin which is also called the home**
- **In addition to its position a turtle also has several other attributes**

Simple Graphics

Heading	**Specified in degrees, the heading or direction increases in value as the turtle turns to the left, or counterclockwise. Conversely, anegative quantity of degrees indicates a right, or clockwise, turn. The turtle is initially facing east, or 0 degrees. North is 90 degrees**
Color	**Initially black, the color can be changed to any of more than 16 million other colors**
Width	**This is the width of the line drawn when the turtle moves. The initial width is 1 pixel. (You'll learn more about pixels shortly.)**
Down	**This attribute, which can be either true or false, controls whether the turtle's pen is up or down. When true (that is, when the pen is down), the turtle draws a line when it moves. When false (that is, when the pen is up), the turtle can move without drawing a line**

- **All these attributes make up a turtle's state**
- **The turtle's state determines how the turtle will behave when any operations are applied to it**
- **A turtle will draw when it is moved if its pen is currently down but it will simply move without drawing when its pen is currently up**

Simple Graphics

- **A turtle is an object its operations are defined as methods**
- **The variable t refers to a Turtle object**
- **The following Table lists the methods of the Turtle class**

Simple Graphics

`Turtle` METHOD	WHAT IT DOES
`t = Turtle()`	Creates a new `Turtle` object and opens its window.
`t.home()`	Moves `t` to the center of the window and then points `t` east.
`t.up()`	Raises `t`'s pen from the drawing surface.
`t.down()`	Lowers `t`'s pen to the drawing surface.
`t.setheading(degrees)`	Points `t` in the indicated direction, which is specified in degrees. East is 0 degrees, north is 90 degrees, west is 180 degrees, and south is 270 degrees.
`t.left(degrees)` `t.right(degrees)`	Rotates `t` to the left or the right by the specified degrees.
`t.goto(x, y)`	Moves `t` to the specified position.
`t.forward(distance)`	Moves `t` the specified distance in the current direction.
`t.pencolor(r, g, b)` `t.pencolor(string)`	Changes the pen color of `t` to the specified RGB value or to the specified string, such as `'red'`. Returns the current color of `t` when the arguments are omitted.
`t.fillcolor(r, g, b)` `t.fillcolor(string)`	Changes the fill color of `t` to the specified RGB value or to the specified string, such as `'red'`. Returns the current fill color of `t` when the arguments are omitted.
`t.begin_fill()` `t.end_fill()`	Enclose a set of turtle commands that will draw a filled shape using the current fill color.

Simple Graphics

`t.width(pixels)`	Changes the width of `t` to the specified number of pixels. Returns `t`'s current width when the argument is omitted.
`t.hideturtle()` `t.showturtle()`	Makes the turtle invisible or visible.
`t.position()`	Returns the current position `(x, y)` of `t`.
`t.heading()`	Returns the current direction of `t`.
`t.isdown()`	Returns `True` if `t`'s pen is down or `False` otherwise.

Simple Graphics

- **An interface should contain all of the information necessary to use an object of a given class**
- **This information includes method headers and documentation about the method's arguments values returned and changes to the state of the associated objects**
- **Python's docstring mechanism allows the programmer to view an interface for an entire class or an individual method by entering expressions of the form help(<class name>) or help(<class name>.<method name>) at a shell prompt**
- **With the use of Turtle object we define a function named drawSquare**
- **This function expects a Turtle object a pair of integers that indicate the coordinates of the square's upper-left corner, and an integer that designates the length of a side**

Simple Graphics

- The function begins by lifting the turtle up and moving it to the square’s corner point
- It then points the turtle due south—270 degrees—and places the turtle’s pen down on the drawing surface
- Finally it moves the turtle the given length and turns it left by 90 degrees four times

```
def drawSquare(t, x, y, length):
    """Draws a square with the given turtle, an upper-left corner point (x, y), and a
    side’s length."""
    t.up()
    t.goto(x, y)
    t.setheading(270)
```

Simple Graphics

```
t.down()
for count in range(4):
    t.forward(length)
    t.left(90)
```

- First an instance of an object's class must be created
- The process of creating an object is called instantiation
- The syntax for instantiating a class and assigning the resulting object to a variable is the following

<variable name> = <class name>(<any arguments>)

Simple Graphics

- **The Turtle class is imported from the turtle module**

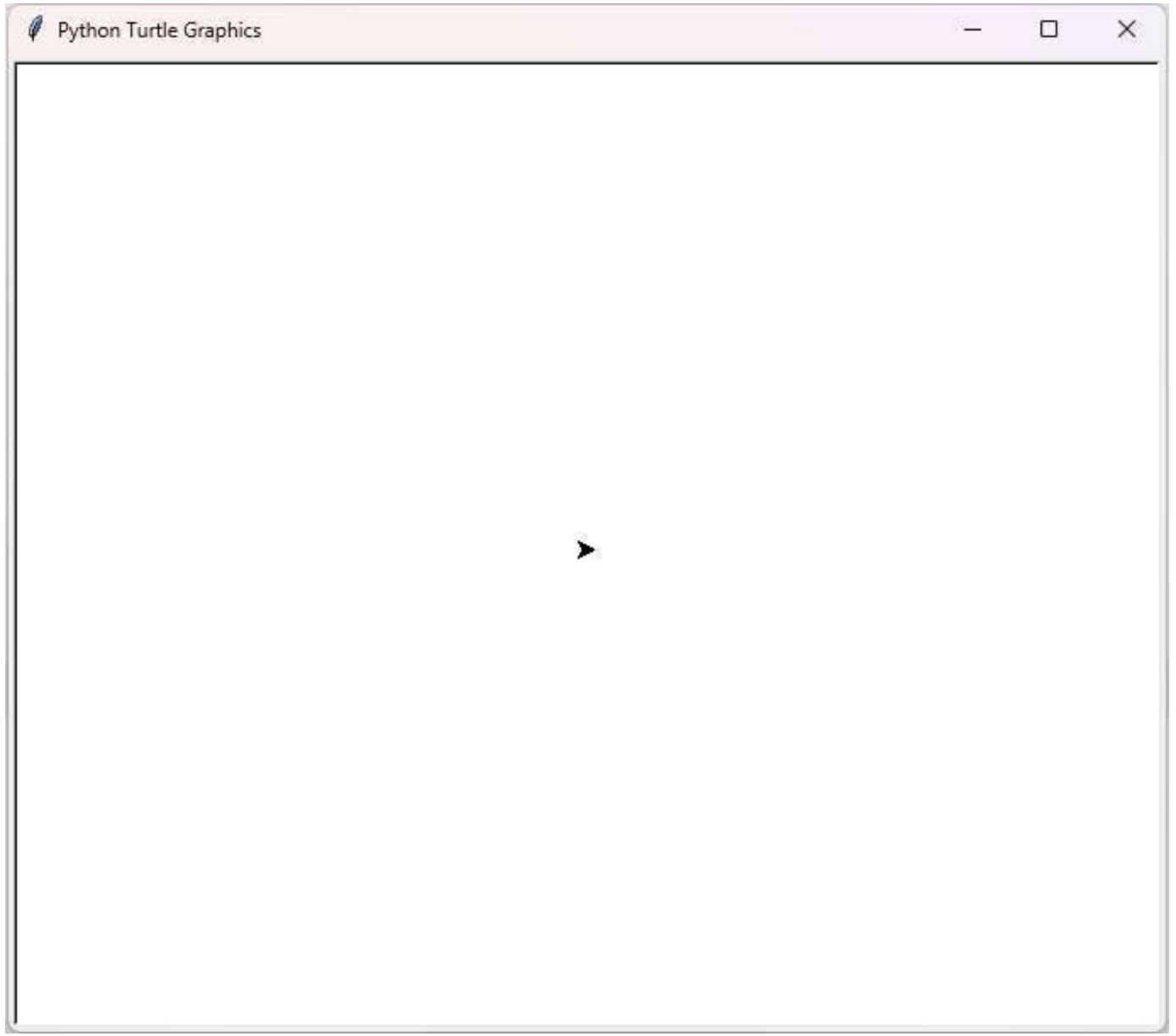
  >>> from turtle import Turtle

- **The next code segment creates and returns a Turtle object**

  >>> t = Turtle()

- **The turtle's icon is located at the home position in the center of the window facing east and ready to draw**

Simple Graphics



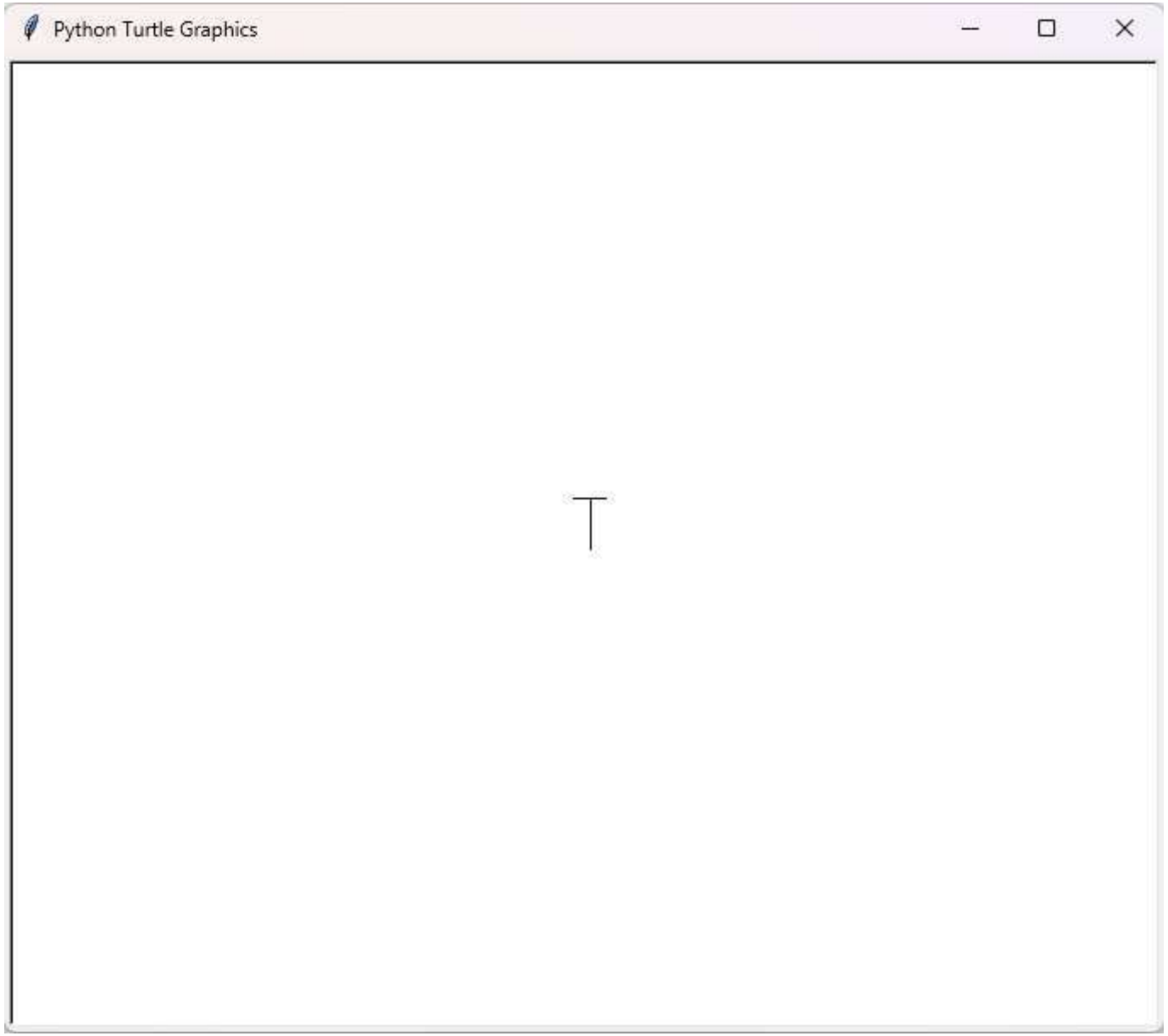
Simple Graphics

- **The purpose of the program is to draw the letter T**
- **It begins at the home position turns 90 degrees left and moves north 30 pixels to draw a vertical line**
- **Then it turns 90 degrees left again to face west picks its pen up, and moves 10 pixels**
- **The turtle next turns to face due east puts its pen down and moves 20 pixels to draw a horizontal line**
- **Finally, we hide the turtle**

Simple Graphics

```
from turtle import Turtle

t = Turtle()

t.left(90)          # Turn to face north

t.forward(30)       # Draw vertical line

t.left(90)          # Turn to face west

t.up()              # Prepare to move without drawing

t.forward(10)       # Move to beginning of horizontal line

t.setheading(0)     # Turn to face east

t.down()            # Prepare to draw

t.forward(20)       # Draw horizontal line

t.hideturtle()      # Make the turtle invisible
```

Simple Graphics



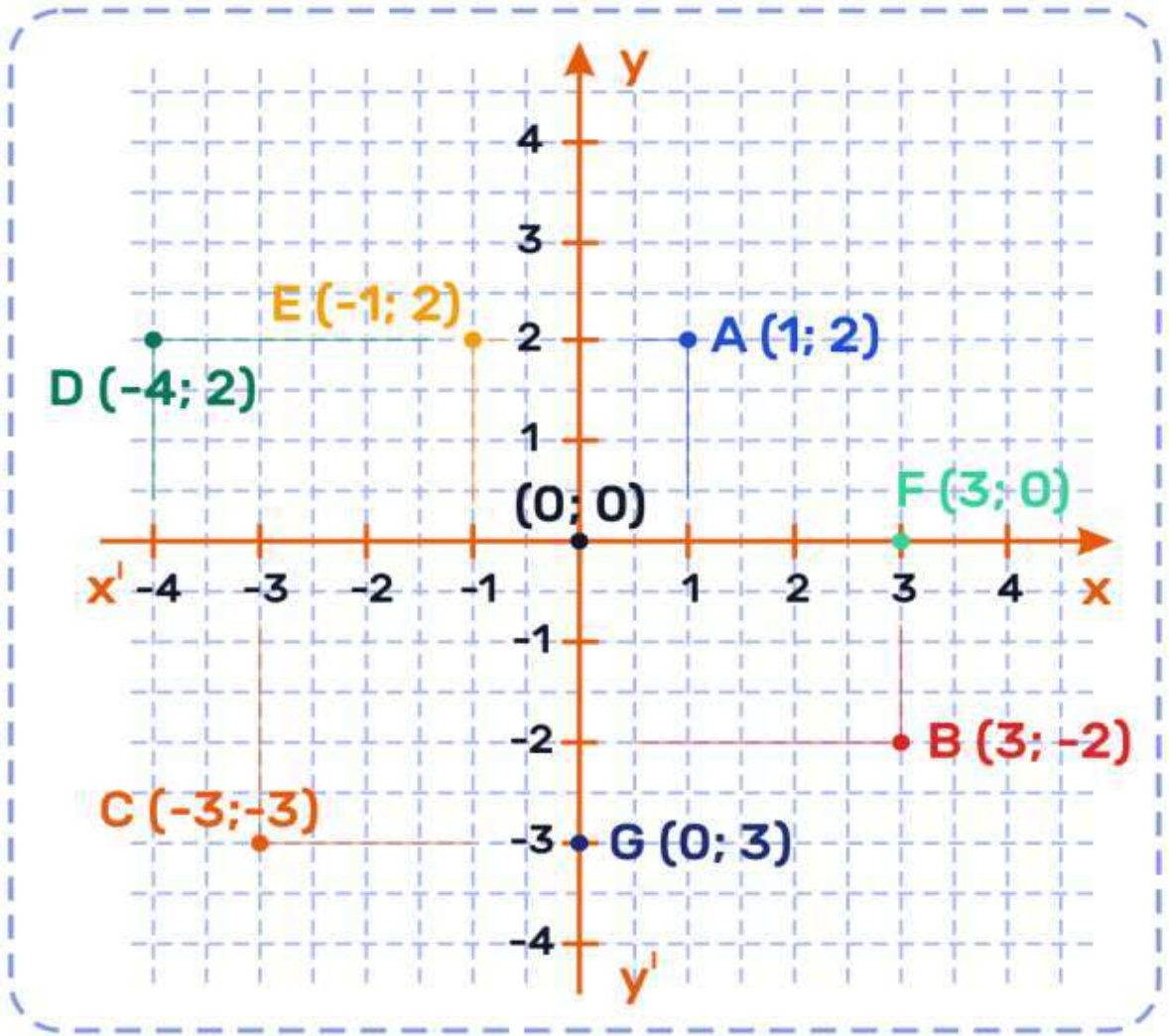
Simple Graphics

- **The purpose of the program is to draw the letter T**
- **It begins at the home position turns 90 degrees left and moves north 30 pixels to draw a vertical line**
- **Then it turns 90 degrees left again to face west picks its pen up, and moves 10 pixels**
- **The turtle next turns to face due east puts its pen down and moves 20 pixels to draw a horizontal line**
- **Finally, we hide the turtle**

Drawing Two-Dimensional Shapes

- **Most of the shapes can be represented as sets of vertices connected by line segments**
- **A triangle has three vertices, a square has four vertices and a pentagon has five vertices**
- **Each vertex is a tuple of coordinates and the set of vertices can be contained in a list**
- **We can define a drawPolygon Python function to draw most two-dimensional shapes**
- **This function expects a Turtle object and a list of at least three vertices as arguments**
- **The function raises the turtle's pen and moves it to the last vertex**
- **The function then lowers the pen and moves the turtle to each vertex in the list starting with the first one**

Cartesian Plane



Brighterly

Drawing Two-Dimensional Shapes

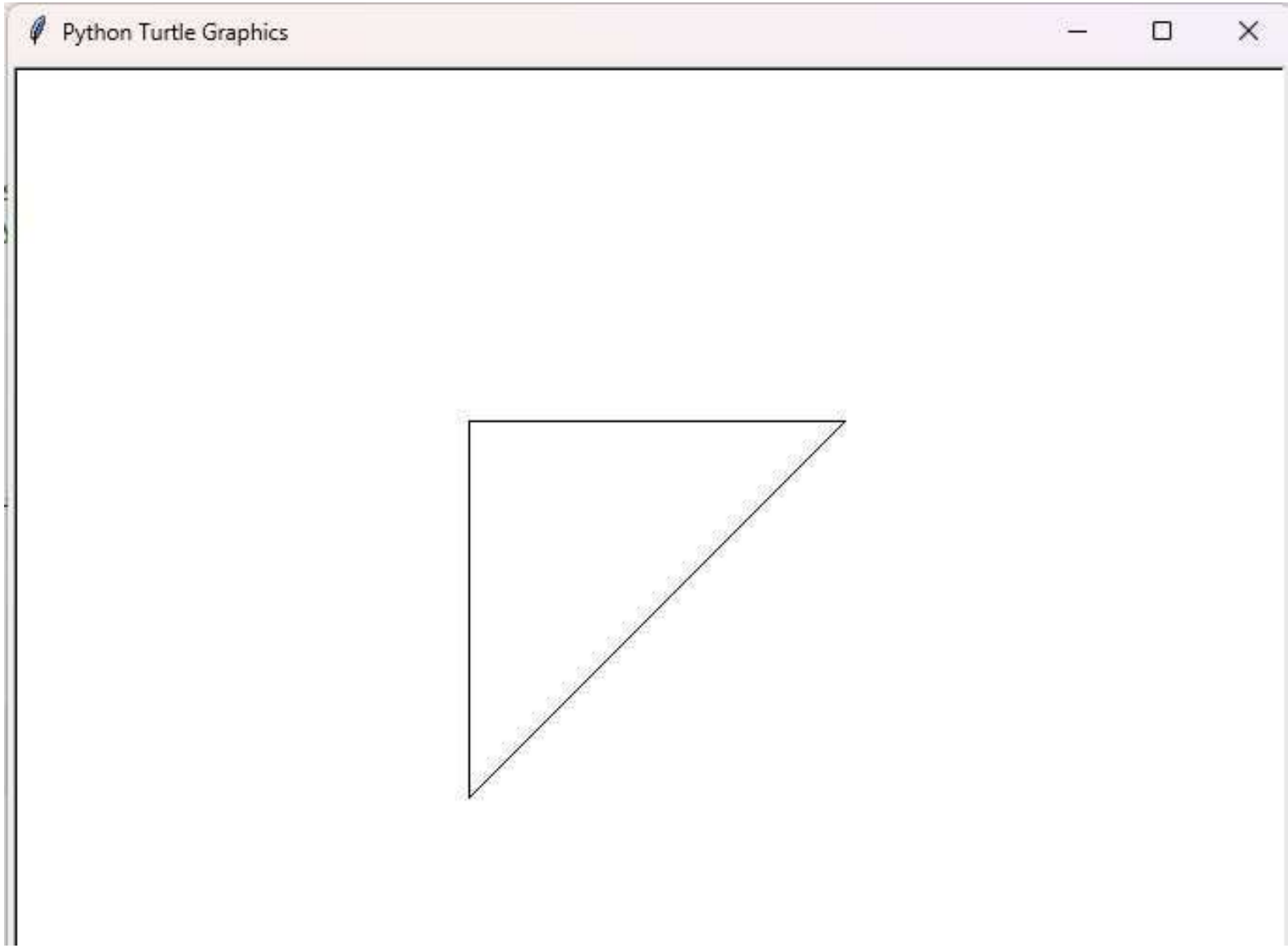
```
from turtle import Turtle

t = Turtle()

t.hideturtle()

def drawPolygon(t, vertices):
    """Draws a polygon from a list of vertices.
    The list has the form [(x1, y1), ..., (xn, yn)]."""
    t.up()
    (x, y) = vertices[-1]
    t.goto(x, y)
```

Drawing Two-Dimensional Shapes

```
t.down()
    for (x, y) in vertices:
        t.goto(x, y)
drawPolygon(t, [(-100, 100),(100,100),(-100,-100)])
```

- The loop traverses a list of tuples so on each pass through the loop the variables x and y in the tuple (x, y) are assigned the corresponding values within the current tuple in the list

Drawing Two-Dimensional Shapes



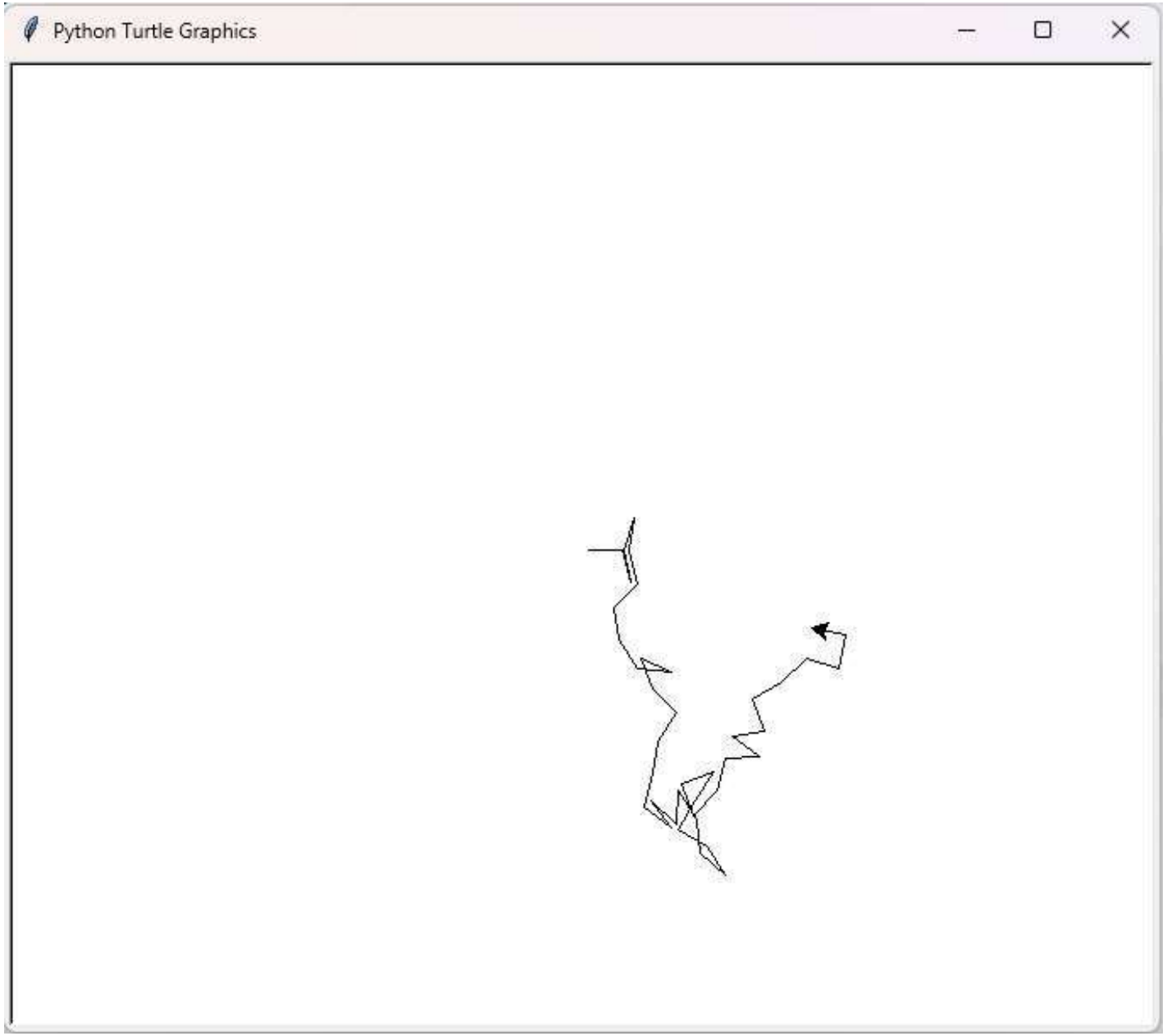
Taking a Random Walk

- **Animals often appear to wander about randomly searching for food, water, shelter and so forth**
- **A turtle wander about randomly**
- **A turtle engages in this harmless activity by repeatedly turning in a random direction and moving a given distance**
- **The following script defines a function randomWalk that expects as arguments a Turtle object the number of turns and distance to move after each turn**
- **The distance argument is optional and defaults to 20 pixels**
- **When called in this script the function performs 40 random turns with a distance of 40 pixels**

Taking a Random Walk

```
from turtle import Turtle

import random

def randomWalk(t, turns, distance = 20):

    """Turns a random number of degrees and moves a given distance for a fixed number

        of turns."""

    for x in range(turns):

        t.left(random.randint(0, 360))

        t.forward(distance)

randomWalk(Turtle(), 40)
```

Taking a Random Walk



Colors and the RGB System

- **RGB stands for the color components red, green, and blue**
- **Each color component can range from 0 through 255**
- **The value 255 represents the maximum saturation of a given color component whereas the value 0 represents the total absence of that component**
- **How many total RGB color values are there ?**
- **All of the possible combinations of three values each of which has 256 possible values, or 256 * 256 * 256, or 16,777,216 distinct color values**
- **N bits of memory can represent 2^N distinct data values**
- **Each color component of an RGB color requires 8 bits so the total number of bits needed to represent a distinct color value is 24**

Colors and the RGB System

COLOR	RGB Value
Black	(0, 0, 0)
Red	(255, 0, 0)
Green	(0, 255, 0)
Blue	(0, 0, 255)
Yellow	(255, 255, 0)
Gray	(127, 127, 127)
White	(255, 255, 255)

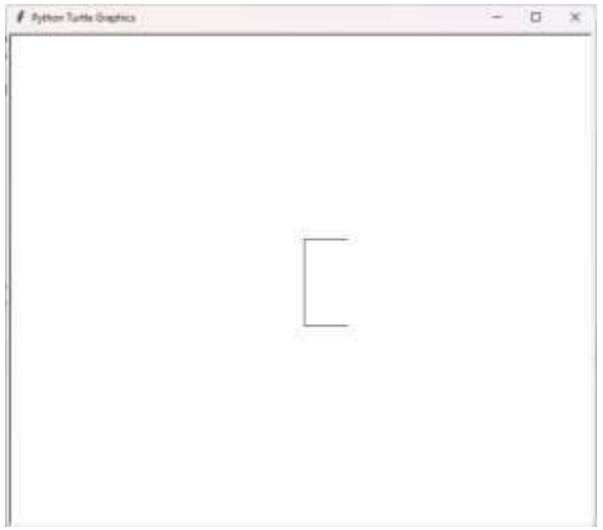
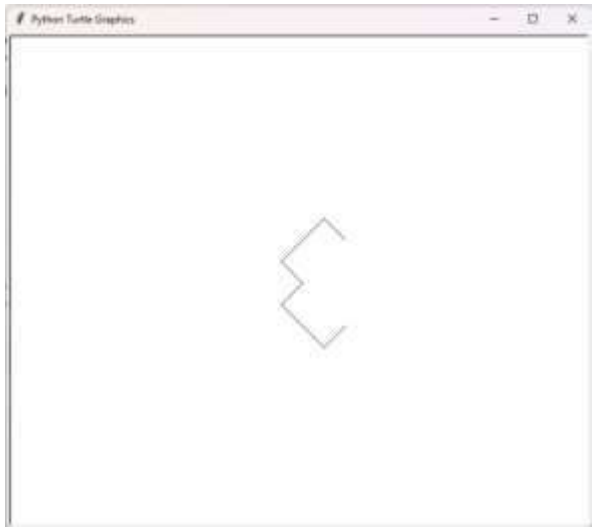
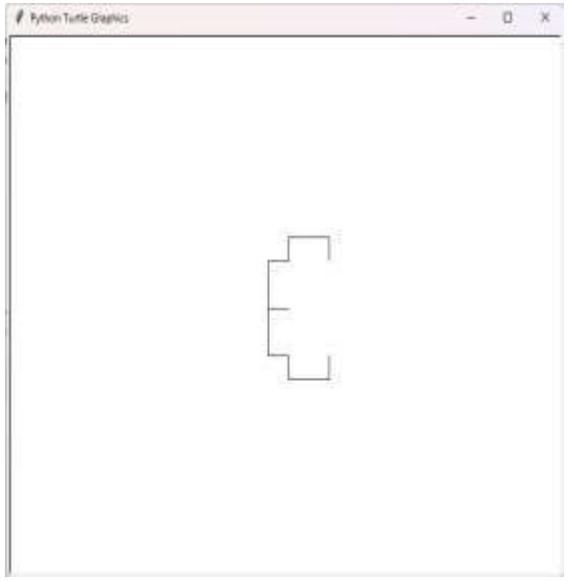
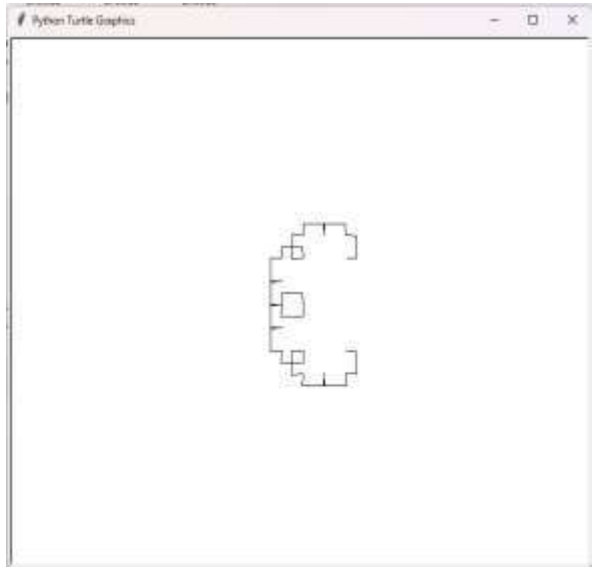
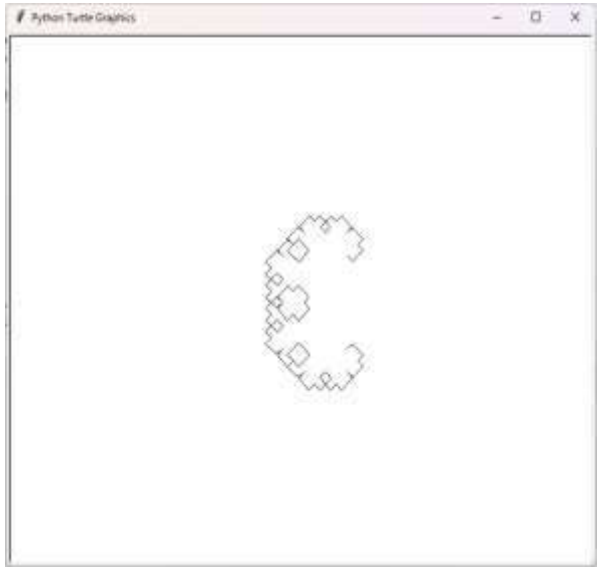
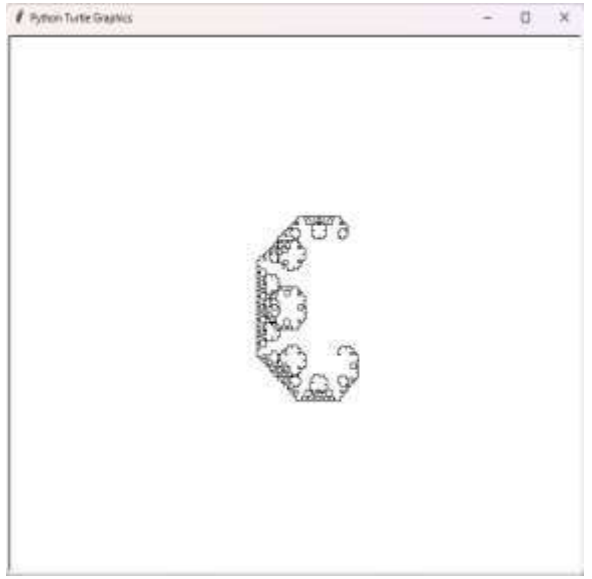
Case Study: Recursive Patterns in Fractals

- **Fractals are highly repetitive or recursive patterns**
- **A fractal curve is not one-dimensional and a fractal surface is not two-dimensional**
- **Instead every fractal shape has its own fractal dimension**
- **An ordinary curve has a precise finite length between any two points**
- **A fractal curve has an indefinite length between any two points**
- **The apparent length of a fractal curve depends on the level of detail in which it is viewed**
- **As you zoom in on a segment of a fractal curve you can see more and more details and its length appears greater and greater**

Case Study: Recursive Patterns in Fractals

- **One example of a fractal curve is the c-curve.**
- **The figure shows the first six levels of c-curves and a level-10 c-curve**
- **The level-0 c-curve is a simple line segment**
- **The level-1 c-curve replaces the level-0 c-curve with two smaller level-0 c-curves that meet at right angles**
- **The level-2 c-curve does the same thing for each of the two line segments in the level-1 c-curve**
- **This pattern of subdivision can continue indefinitely producing quite intricate shapes**
- **In the remainder of this case study we develop an algorithm that uses Turtle graphics to display a c-curve**

Case Study: Recursive Patterns in Fractals

Case Study: Recursive Patterns in Fractals

- **Request :- Write a program that allows the user to draw a particular c-curve in varying degrees**

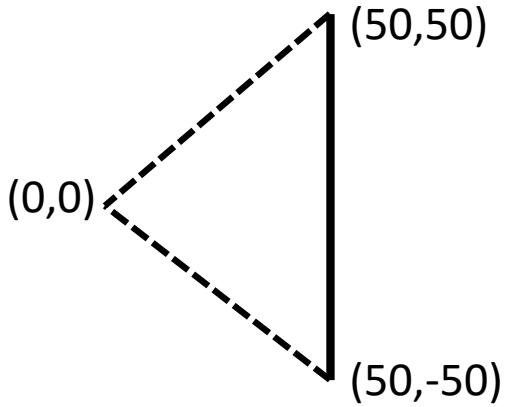
- **Analysis :- The program should prompt the user for the level of the c-curve**
- **After this integer is entered the program should display a Turtle graphics window in which it draws the c-curve**

- **Design :- An N-level c-curve can be drawn with a recursive function**
- **The function receives a Turtle object the end points of a line segment and the current level as arguments**

Case Study: Recursive Patterns in Fractals

- **At level 0 the function draws a simple line segment**
- **A level N c-curve consists of two level N - 1 c-curves constructed as follows**
- **Let xm be (x1 + x2 + y1 - y2) // 2.**
- **Let ym be (x2 + y1 + y2 - x1) // 2.**
- **The first level N - 1 c-curve uses the line segment (x1, y1), (xm, ym), and level N - 1 so the function is called recursively with these arguments**
- **The second level N - 1 c-curve uses the line segment (xm, ym), (x2, y2), and level N - 1, so the function is called recursively with these arguments**
- **For example, in a level-0 c-curve, let (x1, y1) be (50, -50) and (x2, y2) be (50, 50)**

Case Study: Recursive Patterns in Fractals



- **To obtain a level-1 c-curve, use the formulas for computing xm and ym to obtain (xm, ym), which is (0, 0)**
- **Figure shows a solid line segment for the level-0 c-curve and two dashed line segments for the level-1 c-curve that result from these operations**
- **In effect the operations produce two shorter line segments that meet at right angles**
- **The following is the pseudocode for the recursive algorithm**

Case Study: Recursive Patterns in Fractals

```
function cCurve(t, x1, y1, x2, y2, level)
    if level == 0:
        drawLine(x1, y1, x2, y2)
    else
        xm = (x1 + x2 + y1 - y2) // 2
        ym = (x2 + y1 + y2 - x1) // 2
        cCurve(t, x1, y1, xm, ym, level - 1)
        cCurve(t, xm, ym, x2, y2, level - 1)
```

- The function drawLine uses the turtle to draw a line between two given endpoints

Case Study: Recursive Patterns in Fractals

- The program includes the three function definitions of cCurve, drawLine, and main
- The drawLine is an auxiliary function its definition is nested within the definition of cCurve

```
from turtle import Turtle
def cCurve(t, x1, y1, x2, y2, level):
    """Draws a c-curve of the given level."""
    def drawLine(x1, y1, x2, y2):
        """Draws a line segment between the endpoints."""
        t.up()
        t.goto(x1, y1)
        t.down()
        t.goto(x2, y2)
```

Case Study: Recursive Patterns in Fractals

```
  if level == 0:
    drawLine(x1, y1, x2, y2)
else:
    xm = (x1 + x2 + y1 - y2) // 2
    ym = (x2 + y1 + y2 - x1) // 2
    cCurve(t, x1, y1, xm, ym, level - 1)
    cCurve(t, xm, ym, x2, y2, level - 1)
```

Case Study: Recursive Patterns in Fractals

```
def main():
    level = int(input("Enter the level (0 or greater): "))
    t = Turtle()
    t.hideturtle()
    cCurve(t, 50, -50, 50, 50, level)
main()
```

Image Processing

- **The human eye cannot distinguish objects that are closer together than 0.1 mm**
- **A sampling of 10 pixels per millimeter (250 pixels per inch and 62,500 pixels per square inch) would be plenty accurate to distinguish objects**
- **A 3-inch by 5-inch image would need 3 * 5 * 62,500 pixels/inch2 = 937,500 pixels which is approximately one megapixel**
- **Back in the days when disk storage was still expensive computer scientists developed several schemes to compress the data of an image to minimize its file size.**
- **Two of the most popular image file formats are JPEG (Joint Photographic Experts Group) and GIF (Graphic Interchange Format)**
- **Various data compression schemes are used to reduce the file size of a JPEG image**

Image Processing

- **JPEG scheme examines the colors of each pixel's neighbors in the grid and if any color values are the same their positions rather than their values are stored thus potentially saving many bits of storage**
- **Before the image is displayed the original color values are restored during the process of decompression**
- **This scheme is called lossless compression meaning that no information is lost**
- **A GIF image compression algorithm consists of two phases**
- **In the first phase the algorithm analyzes the color samples to build a color palette up to 256 of the most prevalent colors**

Image Processing

- **The algorithm then visits each sample in the grid and replaces it with the key of the closest color in the color palette**
- **The resulting image file thus consists of at most 256 color values and the integer keys of the image's colors in the palette This strategy can potentially save a huge number of bits of storage**
- **The decompression algorithm uses the keys and the color palette to restore the grid of pixels for display**
- **GIF uses a lossy compression scheme it works very well for images with broad flat areas of the same color such as cartoons, backgrounds, and banners**

Image Processing

- **Image manipulation programs either transform the information in the pixels or alter the arrangement of the pixels in the image**
- **These manipulation programs can do the following**
- **Converting a colour Image to Black and White**
- **Converting an Image to Grayscale**
- **Copying an Image**
- **Blurring an Image**
- **Perform edge detection on an image**
- **Reducing the Image Size**

The Properties of Images

- **The coordinates of the pixels in this two-dimensional grid range from (0, 0) at the upper-left corner of an image to (width - 1, height - 1) at the lower-right corner, where width and height are the image's dimensions in pixels**
- **The screen coordinate system for the display of an image is somewhat different from the standard Cartesian coordinate system that we used with Turtle graphics where the origin (0,0) is at the center of the rectangular grid**
- **The RGB color system is a common way of representing the colors in images.**
- **An image consists of a width, a height, and a set of color values accessible by means of (x, y) coordinates**

The Properties of Images

- A color value consists of the tuple (r, g, b) where the variables refer to the integer values of its red, green, and blue components respectively
- For image processing there is a small module in Python called images
- This module allows the programmer to load an image from a file, view the image in a window, examine and manipulate an image's RGB values, and save the image to a file
- The images module includes a class named Image
- The Image class represents an image as a two-dimensional grid of RGB values
- The methods for the Image class are listed in the following Table
- The variable i refers to an instance of the Image class

The Properties of Images

Image METHOD	WHAT IT DOES
`i = Image(filename)`	Loads and returns an image from a file with the given filename. Raises an error if the filename is not found or the file is not a GIF file.
`i = Image(width, height)`	Creates and returns a blank image with the given dimensions. The color of each pixel is white, and the filename is the empty string.
`i.getWidth()`	Returns the width of `i` in pixels.
`i.getHeight()`	Returns the height of `i` in pixels.
`i.getPixel(x, y)`	Returns a tuple of integers representing the RGB values of the pixel at position (x, y).
`i.setPixel(x, y, (r, g, b))`	Replaces the RGB value at the position (x, y) with the RGB value given by the tuple `(r, g, b)`.
`i.draw()`	Displays `i` in a window. The user must close the window to return control to the method's caller.
`i.clone()`	Returns a copy of `i`.
`i.save()`	Saves `i` under its current filename. If `i` does not yet have a filename, `save` does nothing.
`i.save(filename)`	Saves `i` under `filename`. Automatically adds a `.gif` extension if `filename` does not contain it.

The images Module

- **The images module accepts only image files in GIF format**
- **To draw an image enter the following commands**

```
>>> from images import Image
>>> image = Image("apple.gif")
>>> image.draw()
```

- **Once an image has been created the width and height can be examined**

```
>>> image.getWidth()
200
>>> image.getHeight()
150
```

The images Module

- **The image's string representation can be printed by**

  >>> print(image)

  File name: apple.gif

  Width: 200

  Height: 150

- **The method getPixel returns a tuple of the RGB values at the given coordinates**
- **The following session shows the information for the pixel at position (0, 0) which is at the image's upper-left corner**

  >>> image.getPixel(0, 0)

  (221, 206, 173)

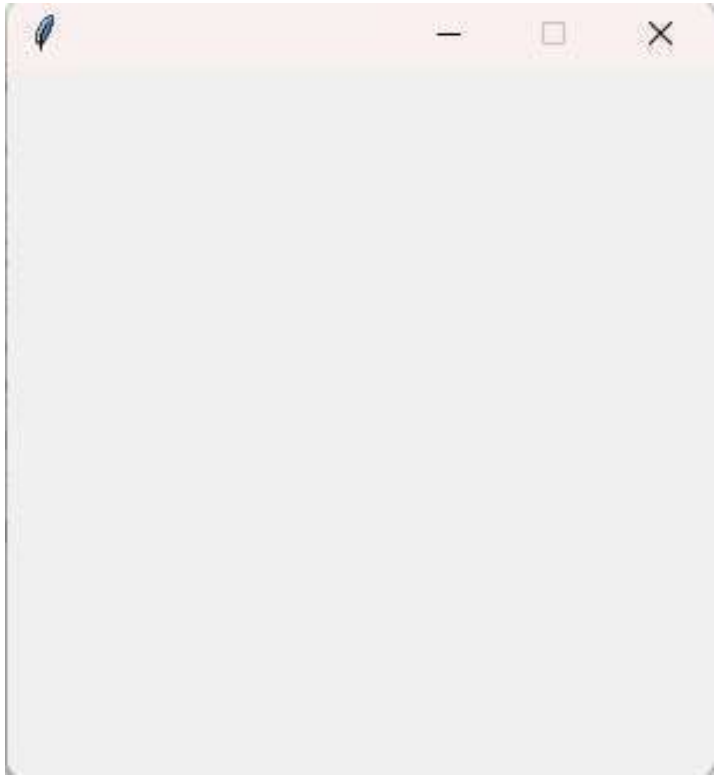
The images Module

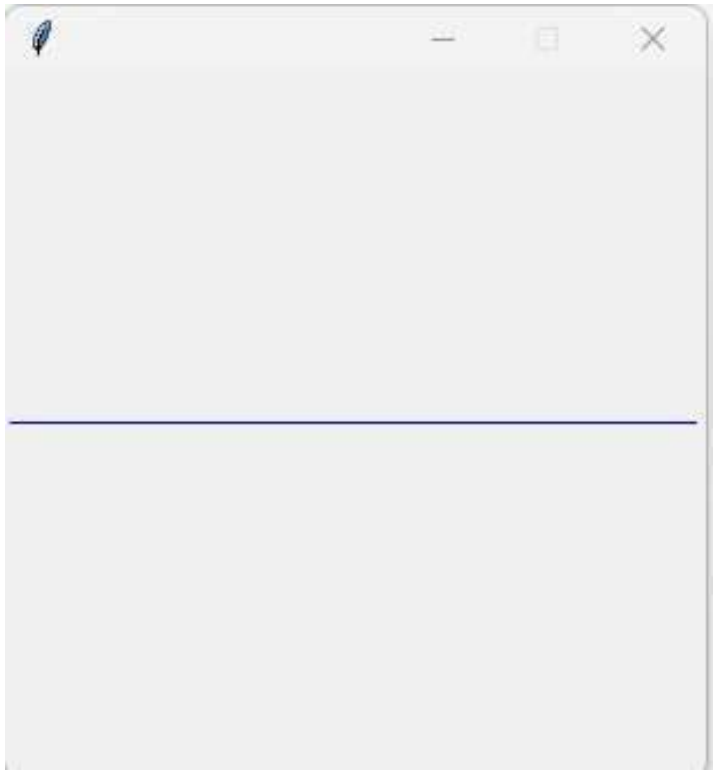
- The method setPixel can be used to replace an RGB value at a given position in an image
- The next program creates a new 300 by 300 image
- The pixels along the middle of the image is replaced with blue pixels
- The loop visits every pixel along the row of pixels whose y coordinate is the image's height divided by 2

```
image = Image(300, 300)
image.draw()
blue = (0, 0, 255)
y = image.getHeight() // 2
```

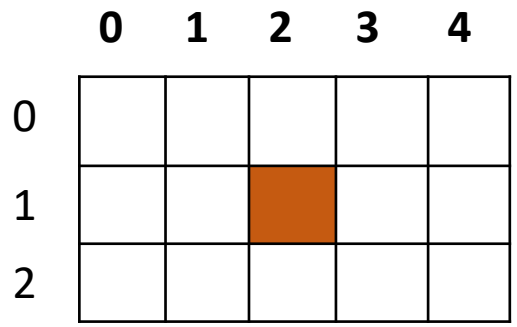
The images Module

```
for x in range(image.getWidth()):
        image.setPixel(x, y, blue)
image.draw()
image.save("horizontal.gif")
```





A Loop Pattern for Traversing a Grid



- **Image processing algorithms use a nested loop structure to traverse a two-dimensional grid of pixels**
- **The grid has a height of 3 rows and a width of 5 columns**
- **Each data value in the grid is accessed with a pair of coordinates using the form (<row>,<column>)**
- **Thus the datum in the middle of the grid which is shaded is at position (1,2)**
- **The datum in the upper-left corner is at the origin of the grid, (0, 0)**

A Loop Pattern for Traversing a Grid

- **A nested loop is used to traverse a grid an outer one to traverse through the row and inner one to traverse through the column**

```
width = 3
height = 5
for x in range(width):
    for y in range(height):
        print((x, y))
    print()
```

A Loop Pattern for Traversing a Grid

- **Many of the algorithms obtain a pixel from the image apply some function to the pixel's RGB values and reset the pixel with the results**
- **As a pixel's RGB values are stored in a tuple manipulating them is quite easy**
- **Suppose we want to increase each of a pixel's RGB values by 10 thereby making the pixel brighter**
- **getPixel can be used to retrieve a tuple and assign it to a tuple that contains three variables (r, g, b) = image.getPixel(x ,y)**
- **The pixel can be reset to new values by resetting the pixel to the tuple**

```
image.setPixel(x, y, (r + 10, g + 10, b + 10))
```

Converting an Image to Black and White

- **A color image can be converted to a black and white image**
- **For each pixel the algorithm computes the average of the red, green, and blue values**
- **The algorithm then resets the pixel's color values to 0 (black) if the average is less than 128(is closer to 0)**
- **Else the algorithm resets the pixel's color values to 255 (white) if the average is greater than 128(is closer to 255)**
- **The code to convert colour image to Black and white is shown**

Converting an Image to Black and White

```
from images import Image

def blackAndWhite(image):
    """Converts the argument image to black and white."""
    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
            (r, g, b) = image.getPixel(x, y)
            average = (r + g + b) // 3
```

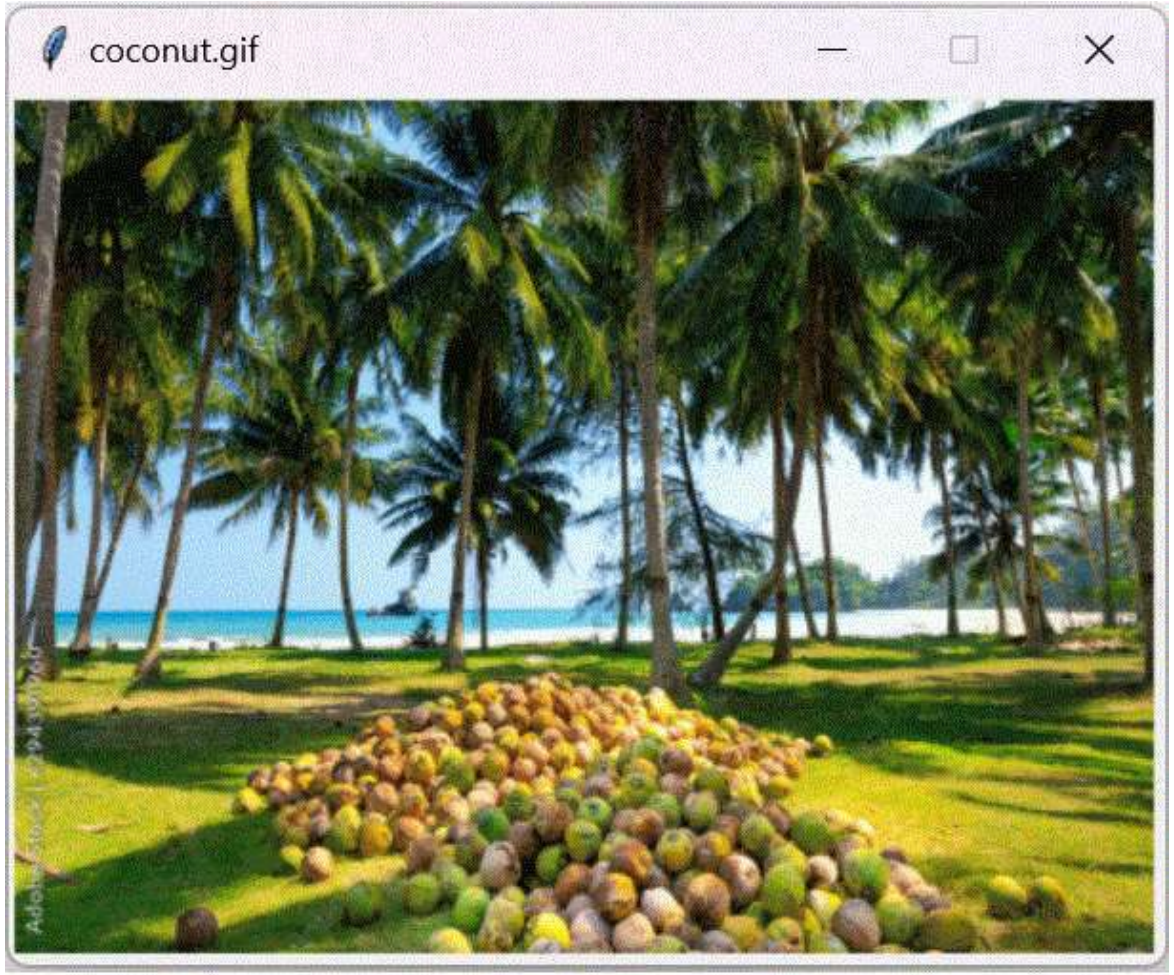
Converting an Image to Black and White

```
if average < 128:
            image.setPixel(x, y, blackPixel)
        else:
            image.setPixel(x, y, whitePixel)


def main(filename = "coconut.gif"):
    image = Image(filename)
    print("Close the image window to continue. ")
    image.draw()
```

Converting an Image to Black and White

```
    blackAndWhite(image)
    print("Close the image window to quit. ")
    image.draw()
main()
```

Converting an Image to Black and White



Converting an Image to Grayscale

- **The human eye is actually more sensitive to green and red than it is to blue**
- **The blue component appears darker than green and red**
- **A scheme that combines the three components needs to take these differences in luminance into account**
- **A more accurate method would weight green more than red and red more than blue**
- **Therefore to obtain the new RGB values instead of adding up the color values and dividing by 3 we should multiply each one by a weight factor and add the results**
- **Psychologists have determined that the relative luminance proportions of green, red, and blue are .587, .299, and .114 respectively**
- **Note that these values add up to 1**

Converting an Image to Grayscale

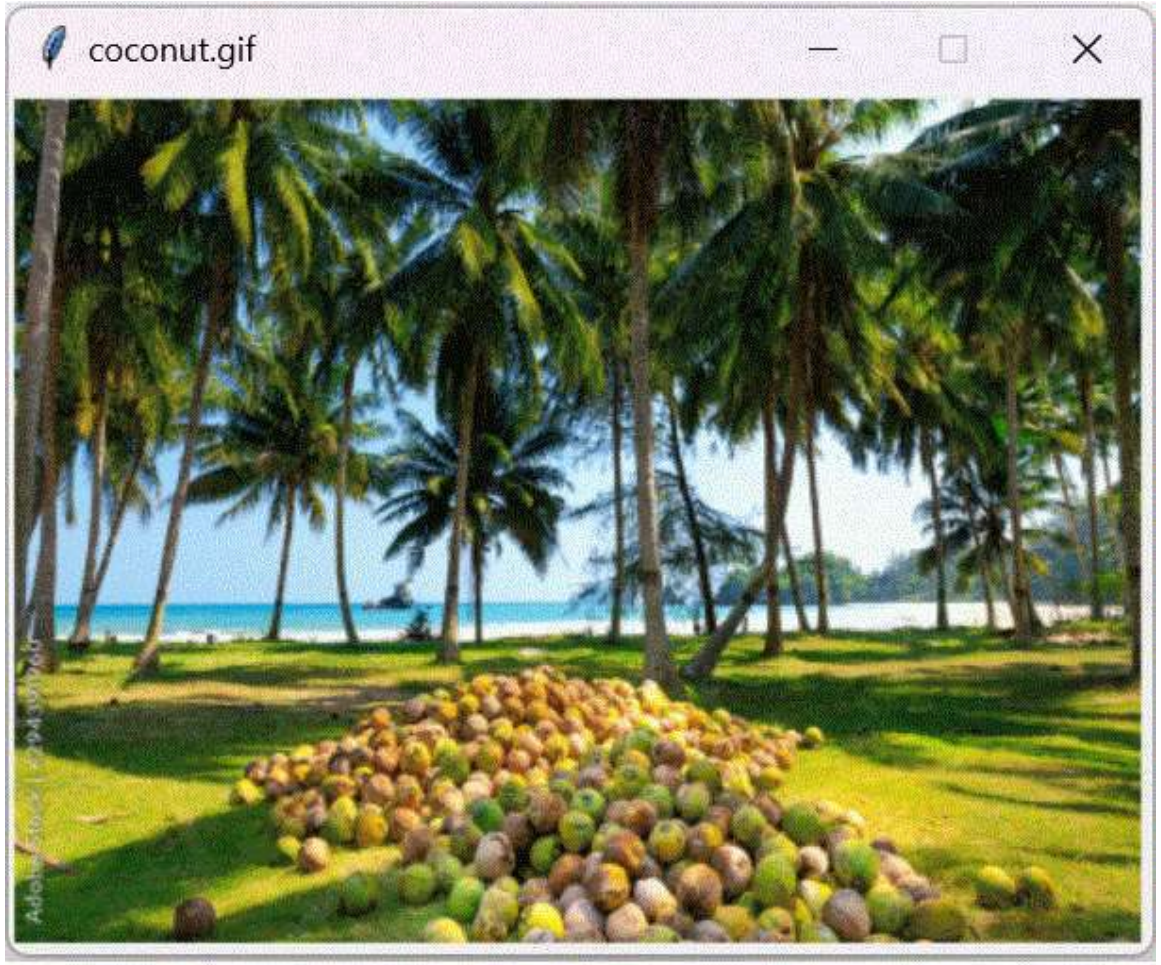
```
from images import Image

def grayscale(image):
    """Converts the argument image to grayscale."""
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
            (r, g, b) = image.getPixel(x, y)
            r = int(r * 0.299)
            g = int(g * 0.587)
            b = int(b * 0.114)
```

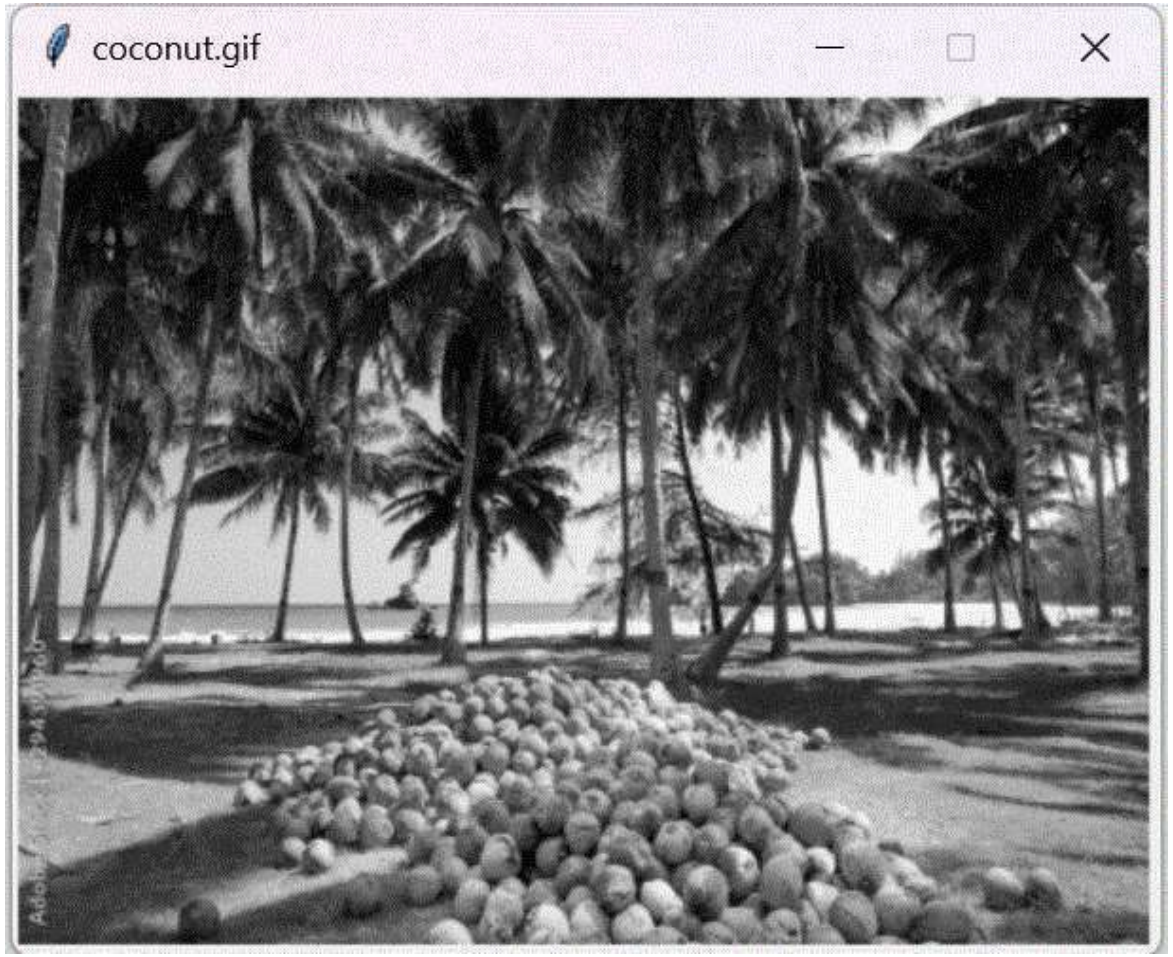
Converting an Image to Grayscale

lum = r + g + b

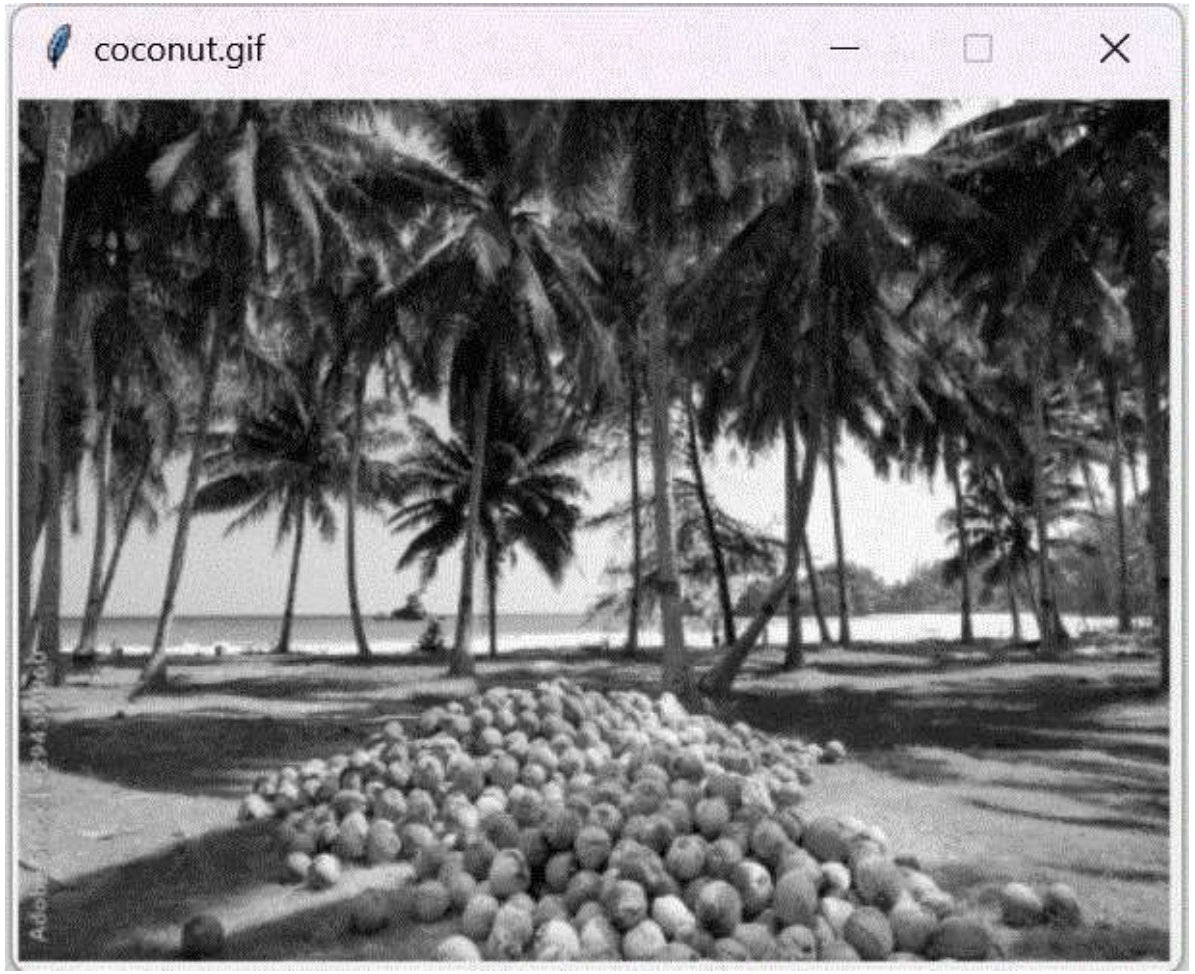
image.setPixel(x, y, (lum, lum, lum))

Converting an Image to Grayscale





Converting an Image to Grayscale



Copying an Image

- **An existing image can be used to generate a brand new image with the desired properties**
- **We can have an exact copy of the original image that retains the same height, width and the pixel information as well**
- **The Image class includes a clone method for this purpose**
- **The method clone builds and returns a new image with the same attributes as the original one**
- **The two images are thus structurally equivalent but not identical**
- **This means that changes to the pixels in one image will have no impact on the pixels in the same positions in the other image**

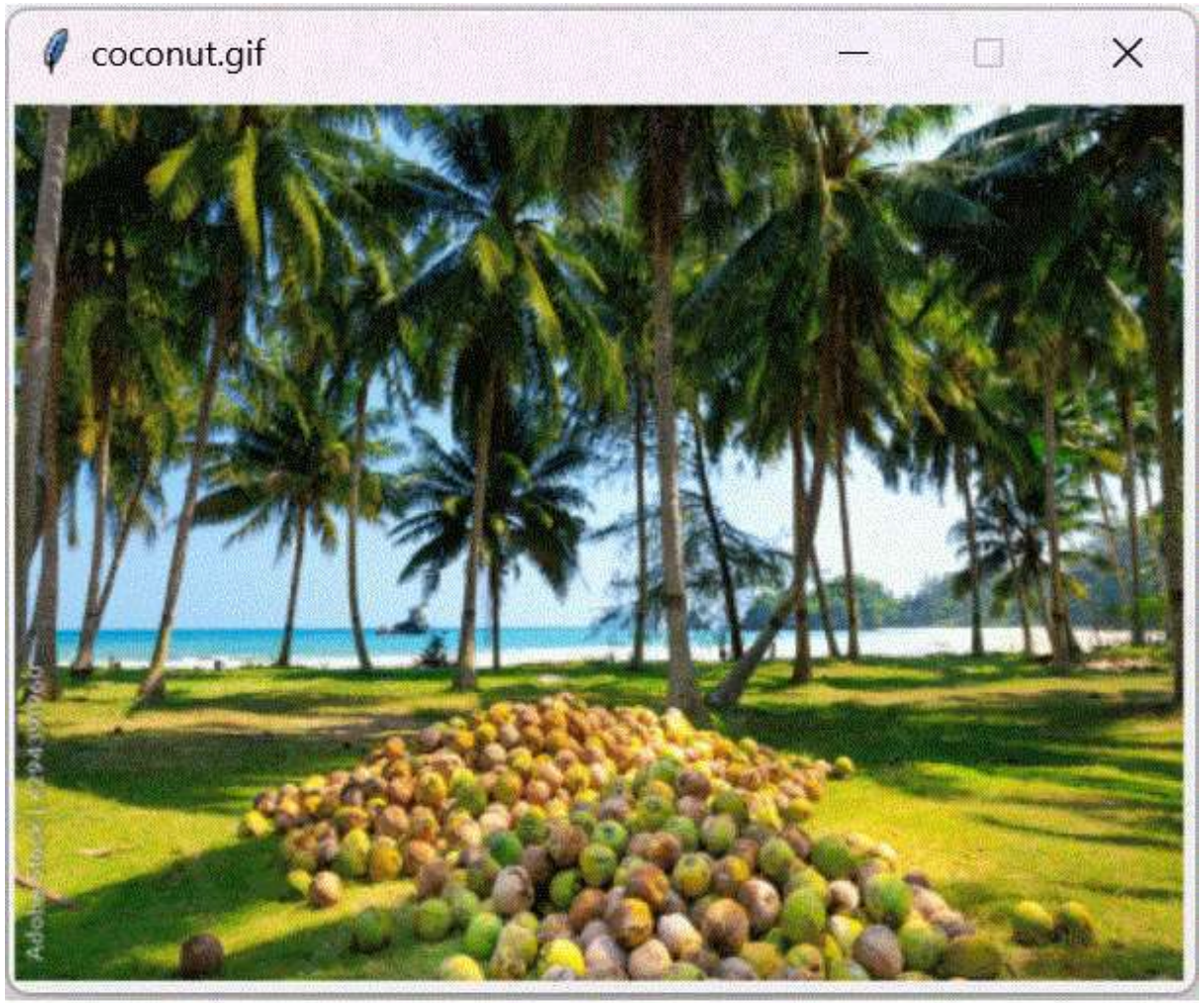
Copying an Image

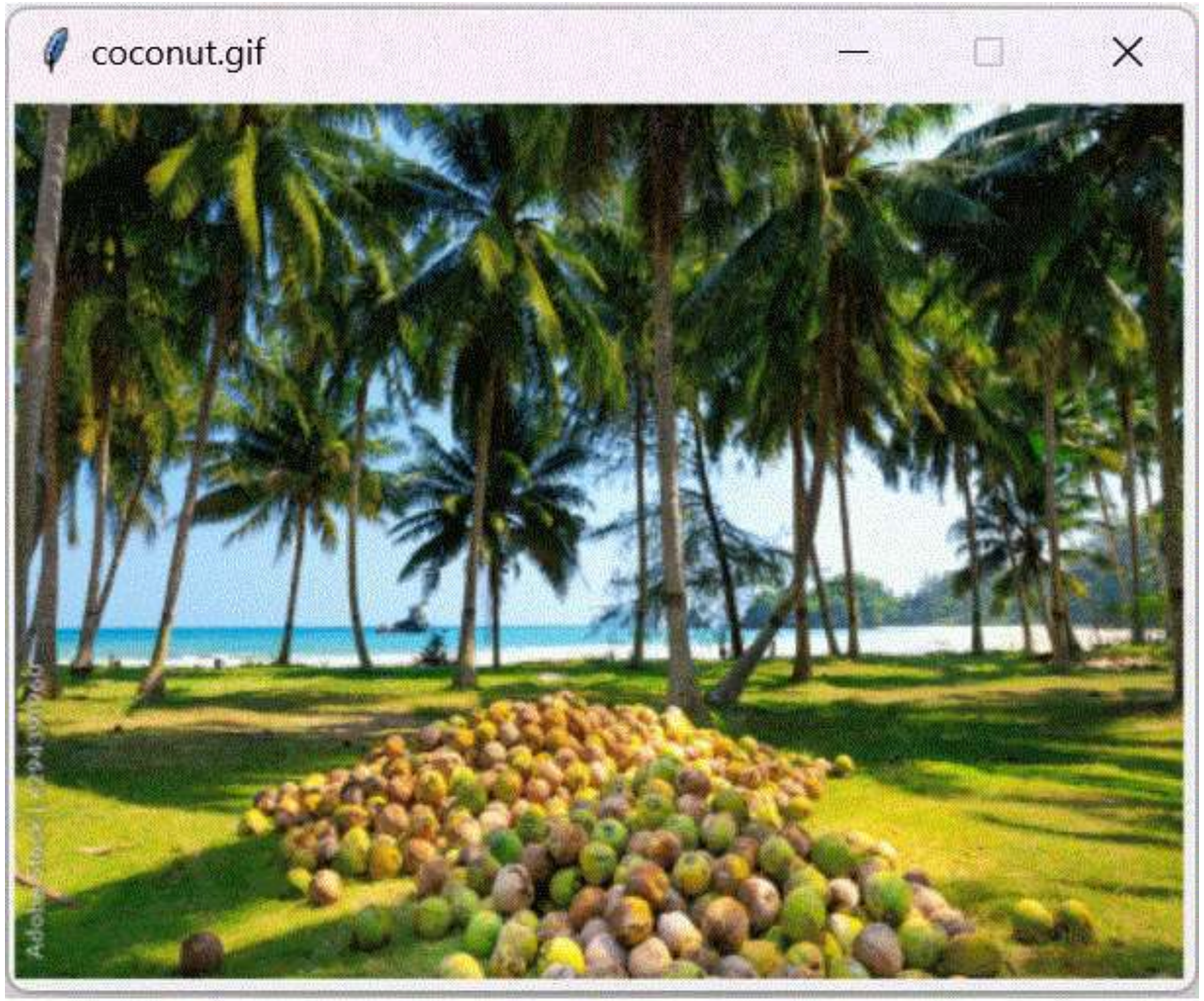
```
from images import Image

def grayscale(image):
    """Converts the argument image to grayscale."""
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
            (r, g, b) = image.getPixel(x, y)
            r = int(r * 0.299)
            g = int(g * 0.587)
            b = int(b * 0.114)
            lum = r + g + b
```

Copying an Image

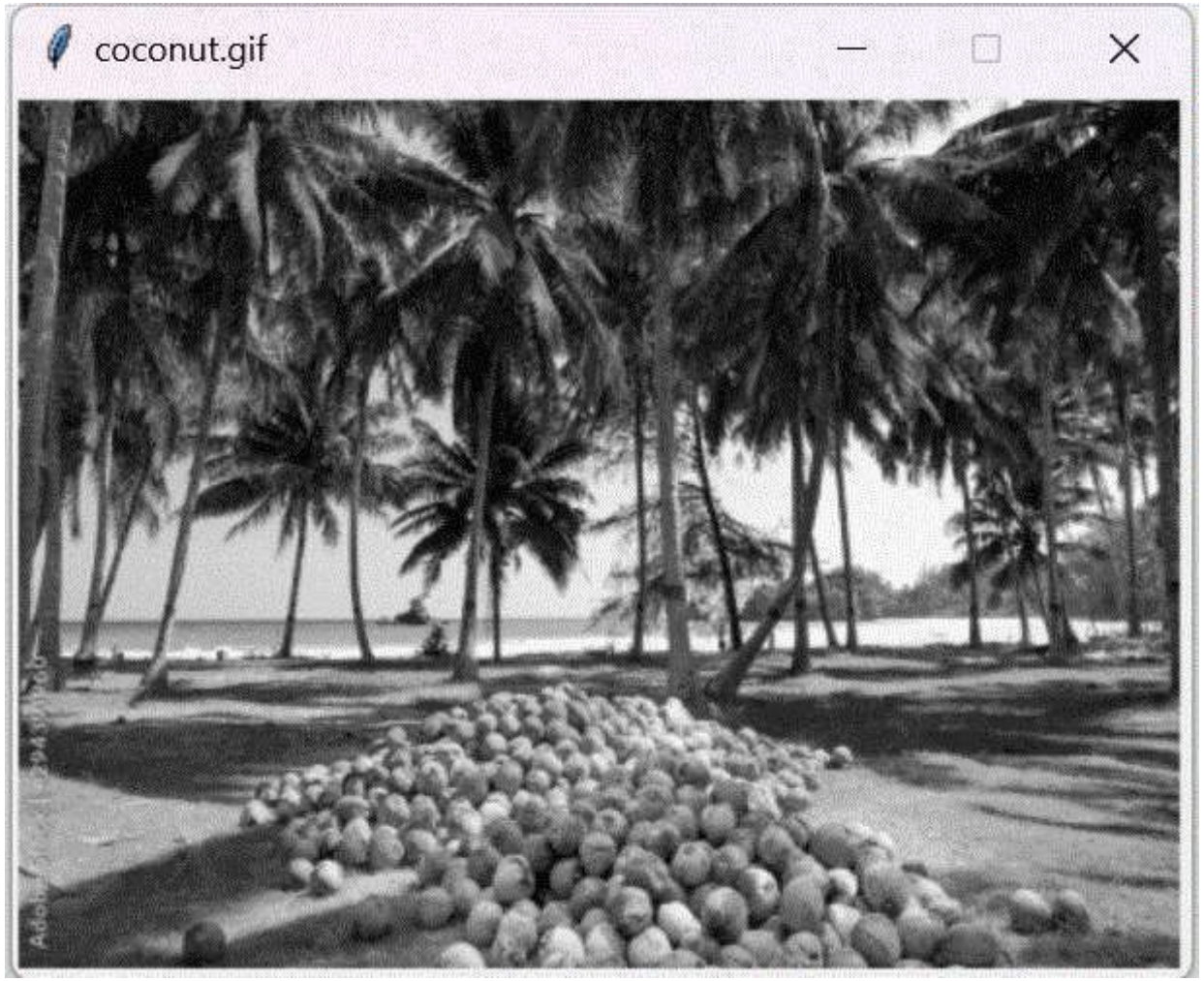
```
        image.setPixel(x, y, (lum, lum, lum))

image = Image(“coconut.gif")

image.draw()

newImage = image.clone()     # Create a copy of image

newImage.draw()

grayscale(newImage)          # Change in second window only

newImage.draw()

image.draw()
```

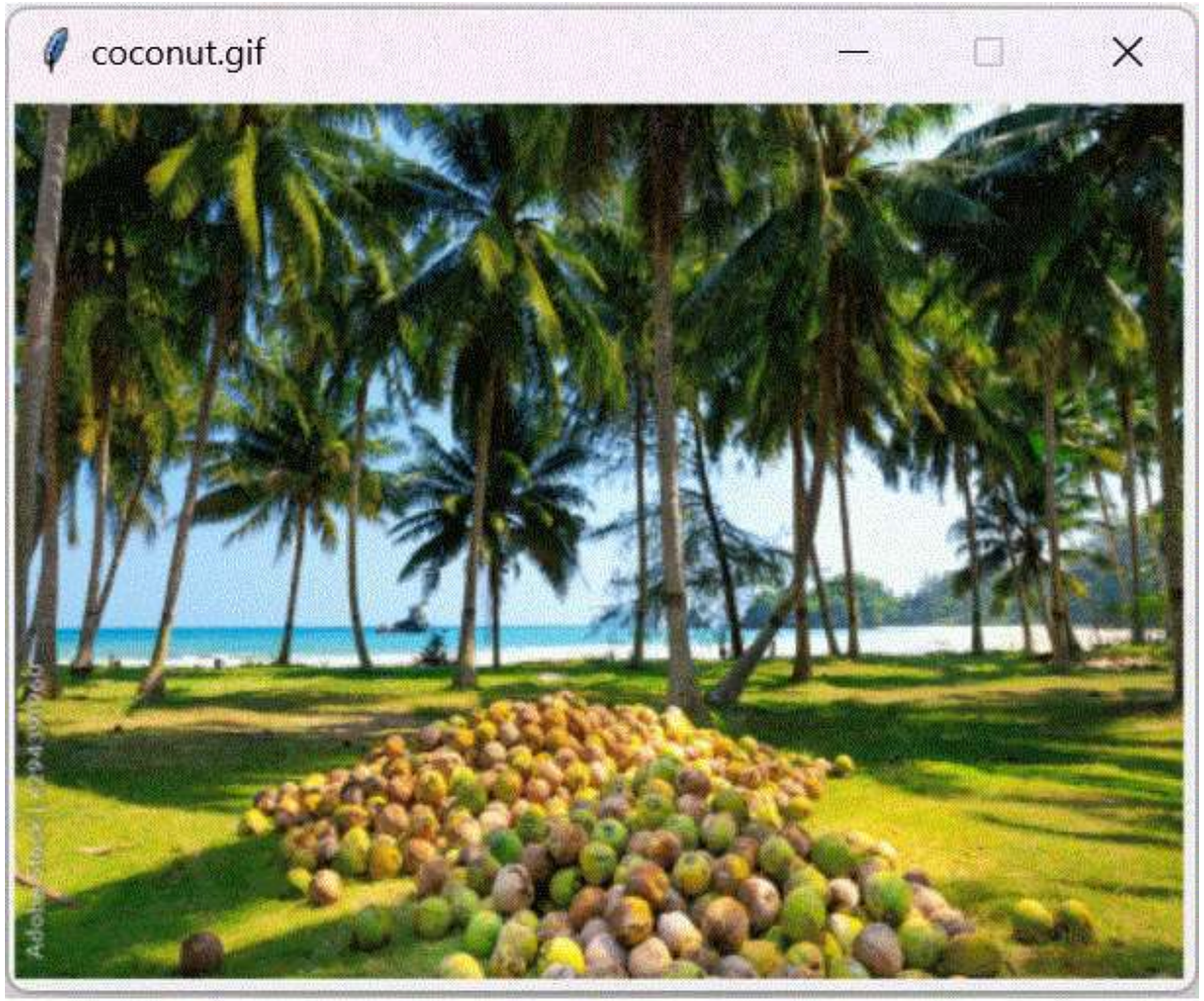
Copying an Image





Copying an Image





Blurring an Image

- **Images contain rough and jagged edges**
- **This can be mitigated by blurring the image's problem areas**
- **Blurring makes these areas appear softer but at the cost of losing some definition**
- **The Blurring algorithm resets each pixel's color to the average of the colors of the four pixels that surround it**
- **The function blur expects an image as an argument and returns a copy of that image with blurring**
- **The function blur begins its traversal of the grid with position (1, 1) and ends with position (width - 2, height - 2)**

Blurring an Image

- The algorithm does not transform the pixels on the image's outer edges because we do not have to check for the grid's boundaries when we obtain information from a pixel's neighbors

```
from images import Image
from functools import reduce
def blur(image):
    """Builds and returns a new image which is a blurred copy of the argument image."""
    def tripleSum(triple1, triple2):          #1
        (r1, g1, b1) = triple1
        (r2, g2, b2) = triple2
```

Blurring an Image

```
return (r1 + r2, g1 + g2, b1 + b2)

    new = image.clone()

    for y in range(1, image.getHeight() - 1):

        for x in range(1, image.getWidth() - 1):

            oldP = image.getPixel(x, y)

            left = image.getPixel(x - 1, y)   # To left

            right = image.getPixel(x + 1, y)  # To right

            top = image.getPixel(x, y - 1)    # Above

            bottom = image.getPixel(x, y + 1) # Below
```

Blurring an Image

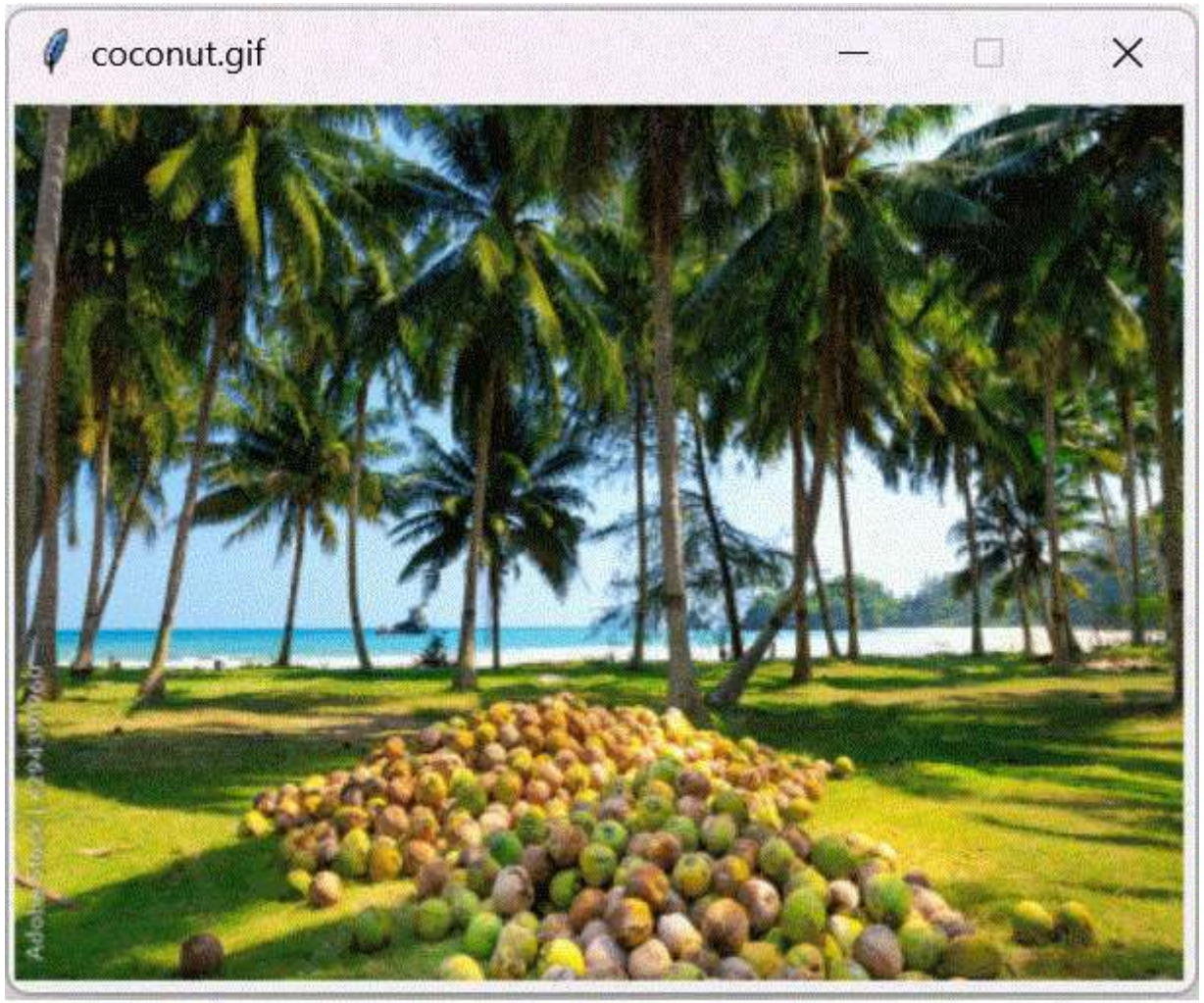
- sums = reduce(tripleSum, [oldP, left, right, top, bottom])

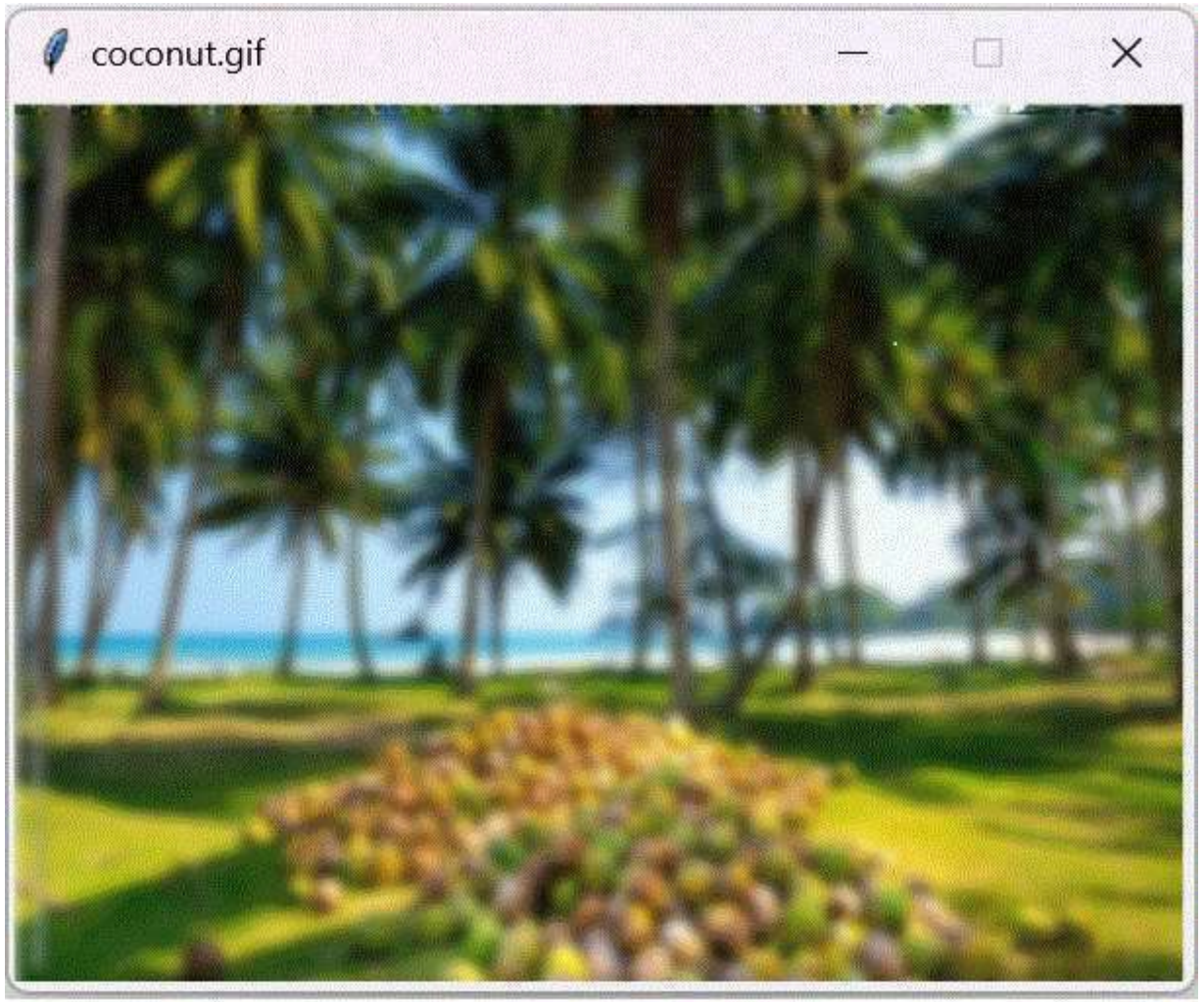
```
        averages = tuple(map(lambda x: x / 5, sums))  #3
        new.setPixel(x, y, averages)
    return new
def main(filename = "apple.gif"):
    image = Image(filename)
    print("Close the image window to continue. ")
    image.draw()
```

Blurring an Image

```
    blur(image)

    print("Close the image window to quit. ")

    image.draw()

main()
```

Blurring an Image





Edge Detection

- **Edge detection removes the full colors of a color image to uncover the outlines of the objects represented in the image**
- **When artists paint pictures they often sketch an outline of the subject in pencil or charcoal and then fill in and color over the outline to complete the painting**
- **A simple edge detection algorithm examines the neighbors below and to the left of each pixel in an image**
- **If the luminance of the pixel differs from that of either of these two neighbors by a significant amount we have detected an edge and we set that pixel's color to black otherwise the pixel's color is set to white**
- **The detectEdges function expects an image and an integer as parameters**

Edge Detection

- **The function returns a new black and white image that explicitly shows the edges in the original image**
- **The integer parameter allows the user to experiment with various differences in luminance**
- **The following figure shows the image before and after detecting edges with luminance thresholds of 10 and 20**

Edge Detection

```
from images import Image
def detectEdges(image, amount):
    """Builds and returns a new image in which the edges of the argument image are
highlighted and the colors are reduced to black and white."""
    def average(triple):
        (r, g, b) = triple
        return (r + g + b) // 3
    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    new = image.clone()
    for y in range(image.getHeight() - 1):
        for x in range(1, image.getWidth()):
            oldPixel = image.getPixel(x, y)
            leftPixel = image.getPixel(x - 1, y)
            bottomPixel = image.getPixel(x, y + 1)
```

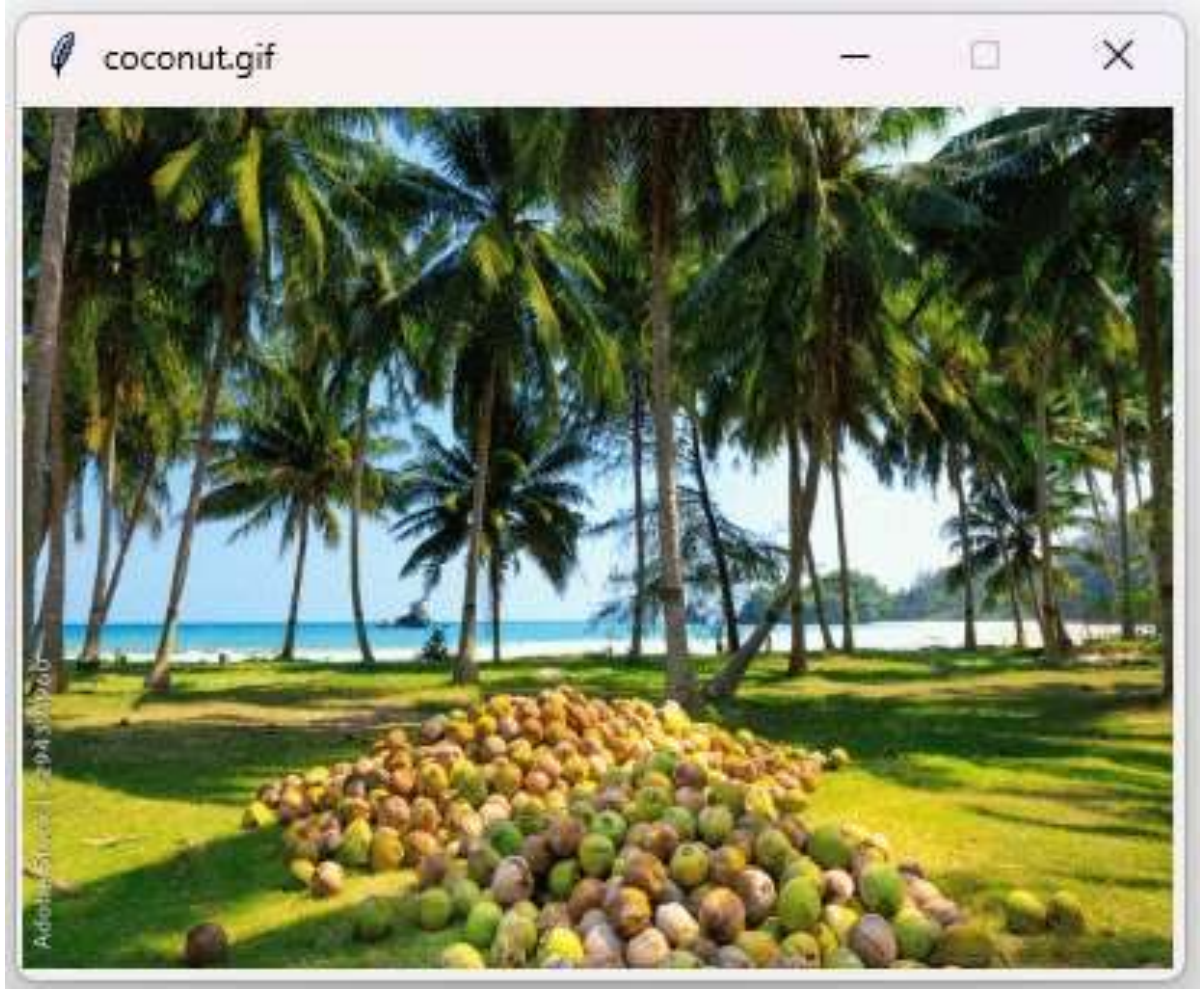
Edge Detection

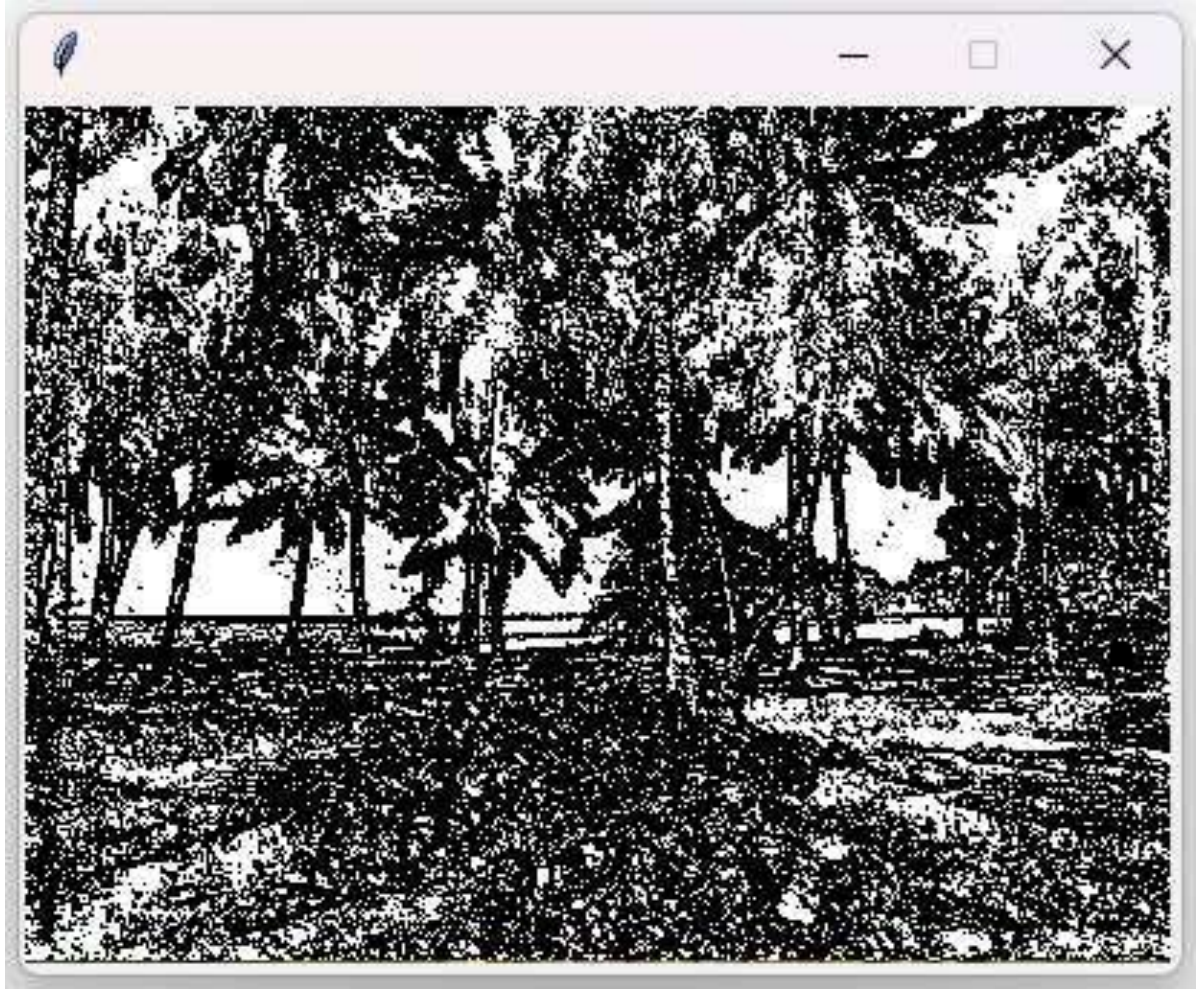
```
oldLum = average(oldPixel)
        leftLum = average(leftPixel)
        bottomLum = average(bottomPixel)
        if abs(oldLum - leftLum) > amount or \
           abs(oldLum - bottomLum) > amount:
            new.setPixel(x, y, blackPixel)
        else:
            new.setPixel(x, y, whitePixel)
    return new

def main(filename = "coconut.gif"):
    image = Image(filename)
    print("Close the image window to continue. ")
    image.draw()
```

Edge Detection

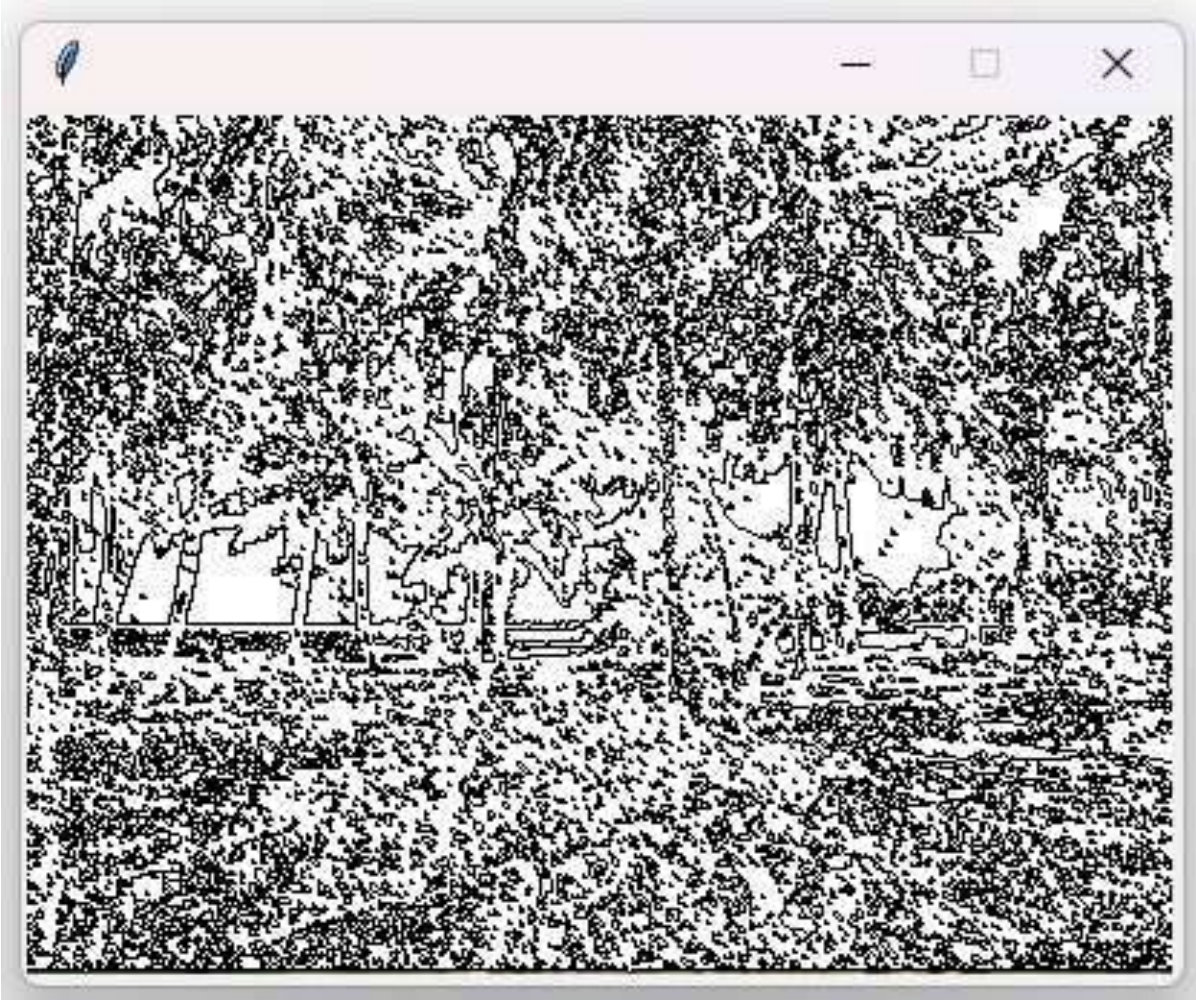
```
    image = detectEdges(image,20)
    print("Close the image window to quit. ")
    image.draw()

main()
```

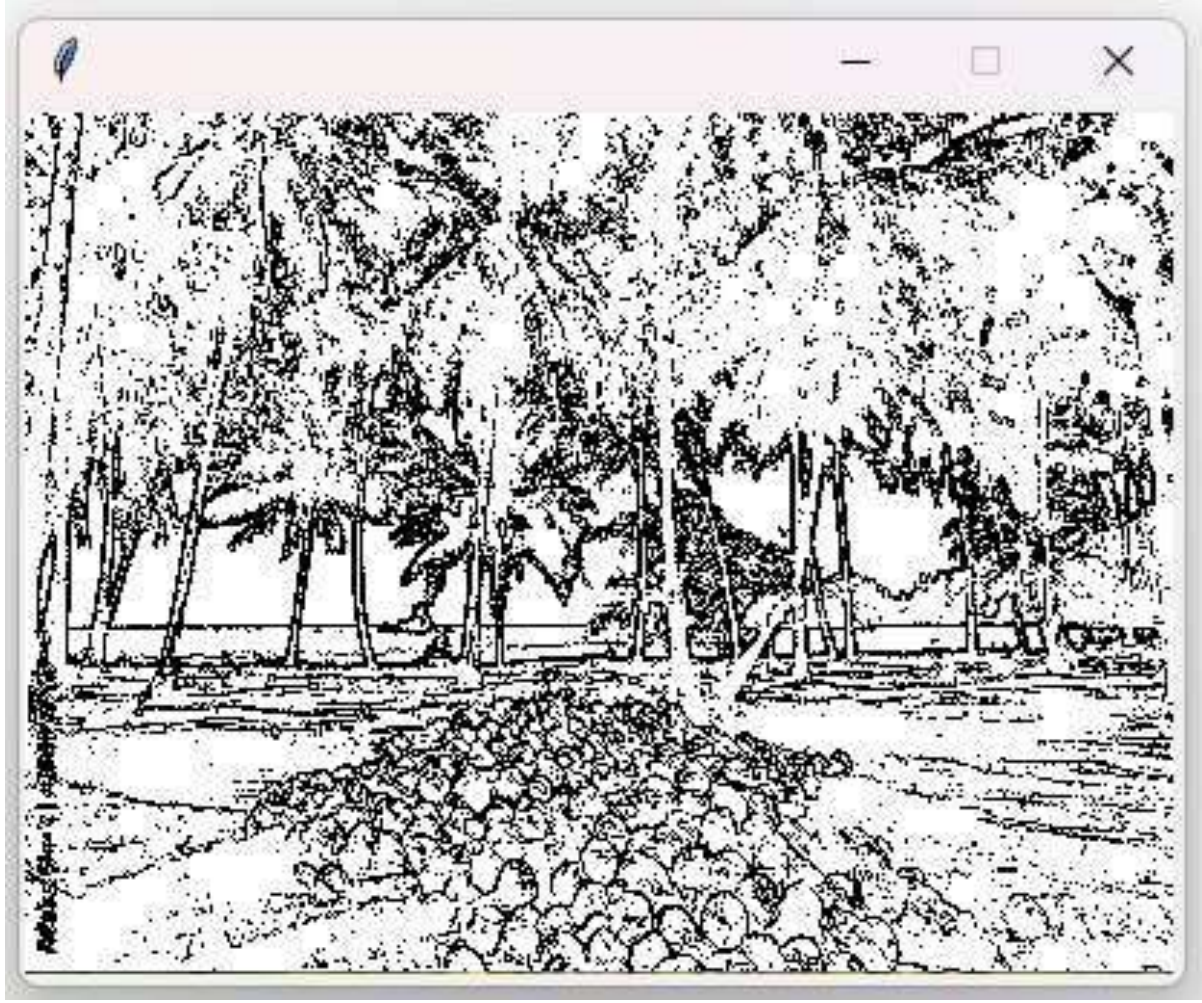
Edge Detection



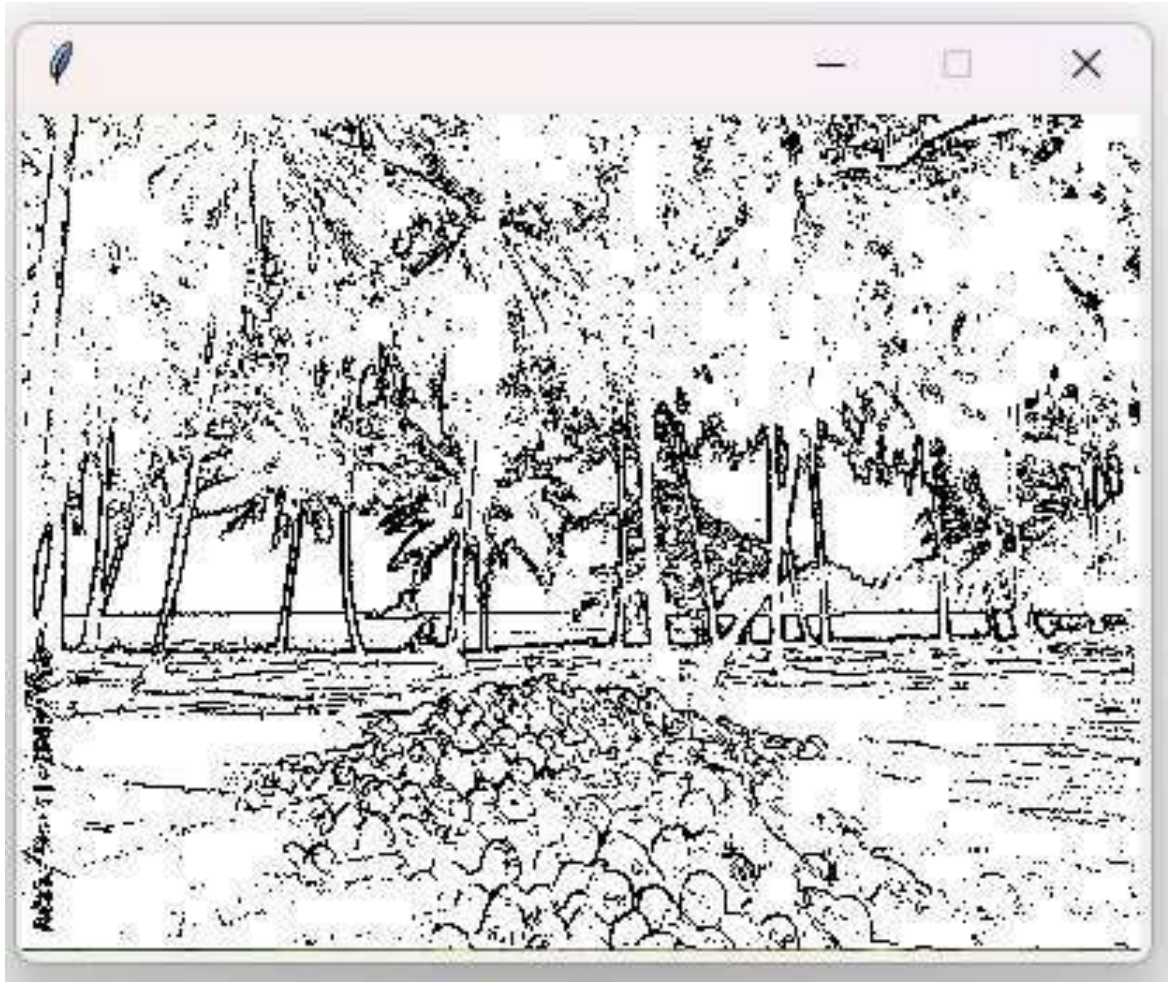


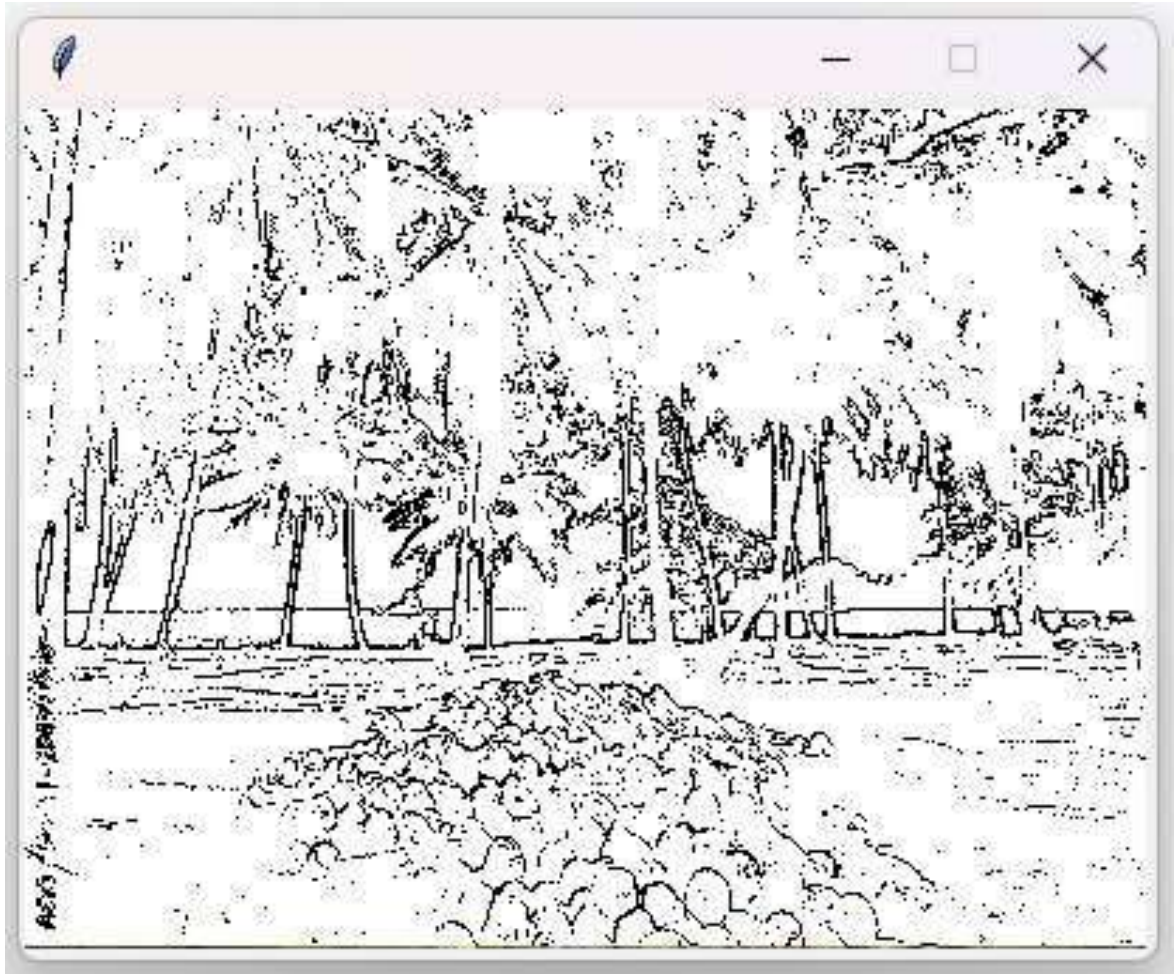
Edge Detection





Edge Detection





Reducing the Image Size

- **The size of an image is its width and height in pixels**
- **Reducing an image's size can dramatically improve its performance characteristics such as load time in a Web page and space occupied on a storage medium**
- **In general if the height and width of an image are each reduced by a factor of N the number of color values in the resulting image is reduced by a factor of N^2**
- **A size reduction usually preserves an image's aspect ratio**
- **Aspect ratio of an image is the ratio of width to height**
- **A simple way to shrink an image is to create a new image whose width and height are a constant fraction of the original image's width and height**

Reducing the Image Size

- **The algorithm to reduce the size of an image by a factor of 2 could copy the color values from every other row and every other column of the original image to the new image**
- **The shrink function expects the original image and a positive integer shrinkage factor as parameters**
- **A shrinkage factor of 2 tells Python to shrink the image to 1⁄2 of its original dimensions and a factor of 3 tells Python to shrink the image to 1⁄3 of its original dimension**
- **The algorithm uses the shrinkage factor to compute the size of the new image and then creates it**

Reducing the Image Size

- **A one-to-one mapping of grid positions in the two images is not possible separate variables are used to track the positions of the pixels in the original image and the new image**
- **The loop traverses the original image and skips positions by incrementing its coordinates by the shrinkage factor**
- **The new image's coordinates are incremented by 1 as usual**
- **Reducing an image's size throws away some of its pixel information**
- **The greater the reduction the greater the information loss**
- **As the image becomes smaller the human eye does not normally notice the loss of visual information and therefore the quality of the image remains stable to perception**

Reducing the Image Size

- **Image enlargement is a tedious process**
- **To increase the size of an image we have to add pixels that were not there**
- **This process can be very complex because we have to transform the existing pixels to blend in with the new ones that are added**
- **As the image gets larger the human eye is in a better position to notice any degradation of quality when comparing it to the original**

Reducing the Image Size

```
from images import Image

def shrink(image, factor):
    """Builds and returns a new image which is a smaller copy of the argument image, by the factor argument."""
    width = image.getWidth()
    height = image.getHeight()
    new = Image(width // factor, height // factor)
    oldY = 0
    newY = 0
```

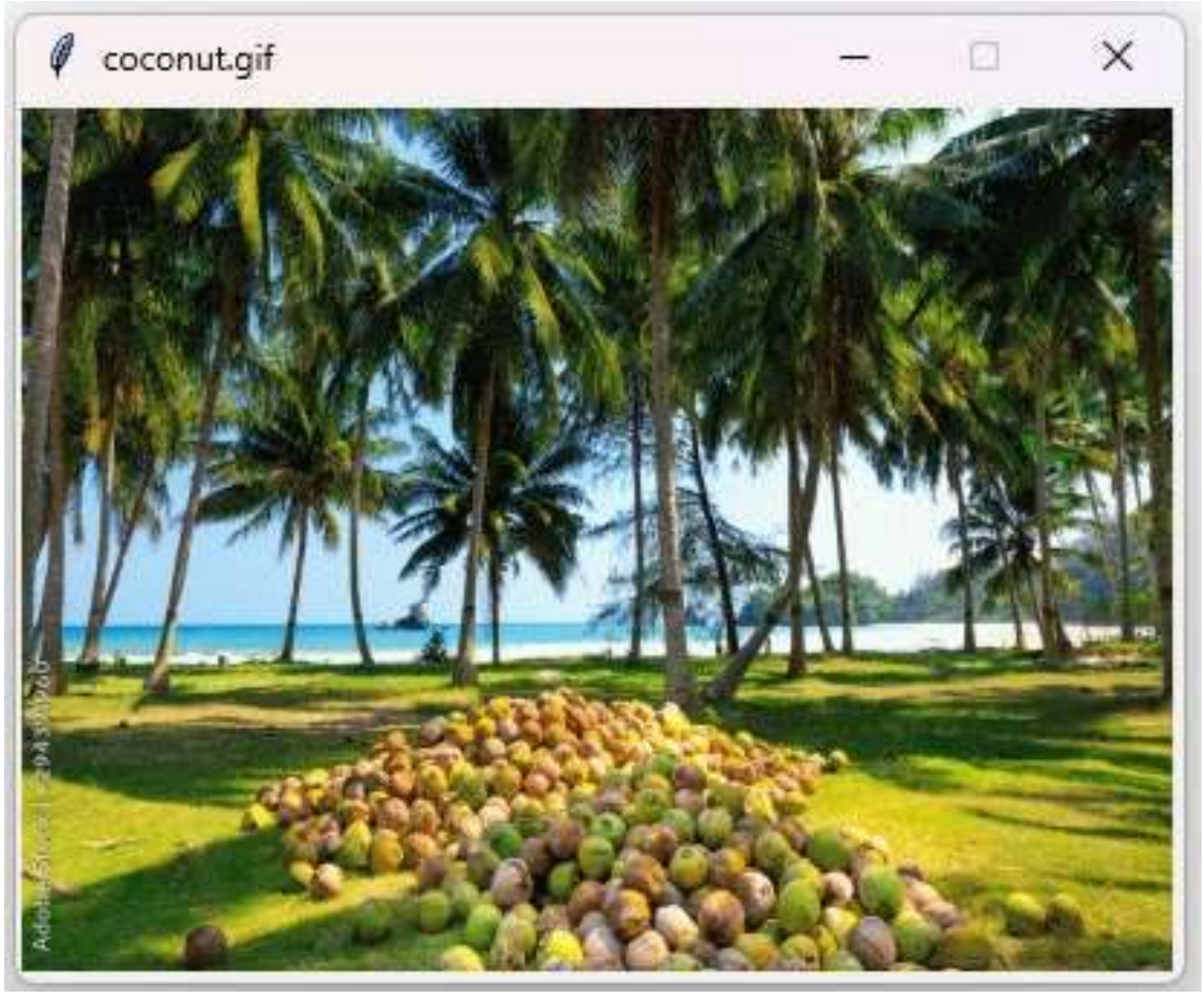
Reducing the Image Size

```
while oldY < height - factor:
        oldX = 0
        newX = 0
        while oldX < width - factor:
            oldP = image.getPixel(oldX, oldY)
            new.setPixel(newX, newY, oldP)
            oldX += factor
            newX += 1
        oldY += factor
        newY += 1
    return new
```

Reducing the Image Size

```
def main(filename = "coconut.gif"):
    image = Image(filename)
    print("Close the image window to continue. ")
    image.draw()
    photos = shrink(image, 3)
    print("Close the image window to quit. ")
    photos.draw()
main()
```

Reducing the Image Size



Module 4
Object Oriented Programming

Object Oriented Programmming

- **Class on instantiation will produce an Object**
- **An object packages a set of data values, its state and a set of operations, its methods in a single entity that can be referenced with a name**
- **A class definition is like a blueprint for each of the objects of that class**
- **A class is a blueprint for creating objects providing initial values for attributes and implementations of methods**
- **Let us look into a simple class definition called Students**

Student Class

- **Each student has a name and a list of test scores**
- **The Student class will allow the user to view a student's name, view a test score at a given position, reset a test score at a given position, view the highest test score, view the average test score, and obtain a string representation of the student's information**
- **When a Student object is created the user supplies the student's name and the number of test scores**
- **Each score is initially presumed to be 0**
- **The interface or set of methods of the Student class is described in the following table**

Student Class

Student METHOD	WHAT IT DOES
`s = Student(name, number)`	Returns a **`Student`** object with the given **`name`** and **`number`** of scores. Each score is initially 0.
`s.getName()`	Returns the student's name.
`s.getScore(i)`	Returns the student's **`i`**th score. **`i`** must range from 1 through the number of scores.
`s.setScore(i, score)`	Resets the student's **`i`**th score to **`score`**. **`i`** must range from 1 through the number of scores.
`s.getAverage()`	Returns the student's average score.
`s.getHighScore()`	Returns the student's highest score.
`s.__str__()`	Same as **`str(s)`**. Returns a string representation of the student's information.

Student Class

The syntax of a class definition is

```
class <class name>(<parent class name>):
    <method definition-1>
    …
    <method definition-n>
```

- **The class definition syntax has two parts a class header and a set of method definitions that follow the class header**
- **The class header consists of the class name and the parent class name**

Student Class

- **Python programmers capitalize the class names to distinguish them from variable names**
- **All Python classes are organized in a tree like class hierarchy**
- **At the top or root of this tree is the most abstract class, named object, which is built in**
- **Each class immediately below another class in the hierarchy is referred to as a subclass whereas the class immediately above it if there is one is called its parent class**
- **If the parenthesized parent class name is omitted from the class definition the new class is automatically made a subclass of object**

Student Class

```
class Student(object):
    """Represents a student."""
    def __init__(self, name, number):
        """Constructor creates a Student with the given name
        and number of scores and sets all scores to 0."""
        self._name = name
        self._scores = []
        for count in range(number):
            self._scores.append(0)
```

Student Class

```
def getName(self):
        """Returns the student's name."""
        return self._name
    def setScore(self, i, score):
        """Resets the ith score, counting from 1."""
        self._scores[i - 1] = score
    def getScore(self, i):
        """Returns the ith score, counting from 1."""
        return self._scores[i - 1]
```

Student Class

```
def getAverage(self):
    """Returns the average score."""
    return sum(self._scores) / len(self._scores)
def getHighScore(self):
    """Returns the highest score."""
    return max(self._scores)
def getLowScore(self):
    """Returns the lowest score."""
    return min(self._scores)
```

Student Class

```
def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name  + "\nScores: " + \
           " ".join(map(str, self._scores))
```

Student Class

```
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20.0
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
```

Docstrings

- **Docstrings occur at three levels**
- **The first level is that of the module**
- **The second level is just after the class header**
- **There might be more than one class defined in a module each class can have a docstring that describes its purpose**
- **The third level is located after each method header**
- **Docstrings at this level serve the same role as they do for function definitions**
- **When we enter help(Student) at a shell prompt the interpreter prints the documentation for the class and all of its methods**

Method Definitions

- **Methods are like functions the syntax of their definitions are similar**
- **Each method definition must include a first parameter named self even if that method seems to expect no arguments when called**
- **When a method is called with an object the interpreter binds the parameter self to that object so that the method's code can refer to the object by name**
- **Methods behave just like functions and can have required or optional arguments and they can return values**

Method Definitions

- **They can create and use temporary variables**
- **A method automatically returns the value None when it includes no return statement**

The __init__ Method and Instance Variables

- The __init__ must begin and end with two consecutive underscores
- This method is also called the class's constructor because it is run automatically when a user instantiates the class

```
def __init__(self, name, number):
    """All scores are initially 0."""
    self._name = name
    self._scores = []
    for count in range(number):
        self._scores.append(0)
```

- Within the class definition the names of instance variables must begin with self

The __init__ Method and Instance Variables

- **s = Student(“Juan”, 5)**
- **When the above code segment is run Python automatically runs the constructor or __init__ method of the Student class**
- **The purpose of the constructor is to initialize an individual object’s attributes**
- **In addition to self the Student constructor expects two arguments that provide the initial values for these attributes**
- **When a reference to a class’s constructor is made it means the __init__ method**
- **The attributes of an object are represented as instance variables**

The __init__ Method and Instance Variables

- Each individual object has its own set of instance variables
- These variables serve as storage for its state
- The scope of an instance variable is the entire class definition
- Within the class definition the names of instance variables must begin with self
- The instance variables self._name and self._scores are initialized to a string and a list respectively
- Python programmers are encouraged to begin the part of an instance variable's name following the dot with a single underscore as in self._name

The __str__ Method

- The __str__ method builds and returns a string representation of an object's state
- When the str function is called with an object that object's __str__ method is automatically invoked to obtain the string that str returns
- The function call str(s) is equivalent to the method call s.__str__() and is simpler to write
- The function call print(s) also automatically runs str(s) to obtain the object's string representation for output
- The code for the __str__ method in the Student class is

The __str__ Method

```
def __str__(self):
    """Returns the string representation of the student."""
    return "Name: " + self._name  + "\nScores: " + \
           " ".join(map(str, self._scores))
```

Accessors and Mutators

- **Methods that allow a user to observe but not change the state of an object are called accessors**
- **Methods that allow a user to modify an object's state are called mutators**
- **The Student class has only one mutator method setScore**
- **The setScore allows the user to reset a test score at a given position**
- **All other methods are accessor methods**
- **The code for the mutator method setScore is as follows**

Accessors and Mutators

```
def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self._scores[i - 1] = score
```

Rational Numbers

- **A rational number consists of two integer parts a numerator and a denominator and is written using the format numerator / denominator**
- **Examples are 1/2, 1/3 and 1/5**
- **Python has no built-in type for rational numbers so we develop a new class named Rational to support this type of data**
- **The interface of the Rational class includes a constructor for creating a rational number an str function for obtaining a string representation and accessors for the numerator and denominator**

Rational Numbers

```
class Rational(object):

    """Represents a rational number."""

    def __init__(self, numer, denom):

        """Constructor creates a number with the given numerator

        and denominator and reduces it to lowest terms."""

        self._numer = numer

        self._denom = denom

        self._reduce()
```

Rational Numbers

```
def numerator(self):
    """Returns the numerator."""
    return self._numer
def denominator(self):
    """Returns the denominator."""
    return self._denom
def __str__(self):
    """Returns the string representation of the number."""
    return str(self._numer) + "/" + str(self._denom)
```

Rational Numbers

```
def _reduce(self):
        """Helper to reduce the number to lowest terms."""
        divisor = self._gcd(self._numer, self._denom)
        self._numer = self._numer // divisor
        self._denom = self._denom // divisor
    def _gcd(self, a, b):
        """Euclid's algorithm for greatest common divisor."""
        (a, b) = (max(a, b), min(a, b))
        while b > 0:
            (a, b) = (b, a % b)
        return a
```

Rational Numbers

```
def __add__(self, other):
    """Returns the sum of the numbers."""
    #Self is the left operand and other is the right operand
    newNumer = self._numer * other._denom + \
        other._numer * self._denom
    newDenom = self._denom * other._denom
    return Rational(newNumer, newDenom)
```

Rational Numbers

```
def __sub__(self, other):
    """Returns the sum of the numbers."""
    #Self is the left operand and other is the right operand
    newNumer = self._numer * other._denom - \
        other._numer * self._denom
    newDenom = self._denom * other._denom
    return Rational(newNumer, newDenom)
```

Rational Numbers

```
def __lt__(self, other):
    """Compares two rational numbers, self and other, using
<."""
    extremes = self._numer * other._denom
    means = other._numer * self._denom
    return extremes < means
```

Rational Numbers

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                   # Object identity?
        return True
    elif type(self) != type(other):     # Types match?
        return False
    else:
        return self._numer == other._numer and \
            self._denom == other._denom
```

Rational Numbers

```
>>> oneHalf = Rational(1, 2)

>>> oneSixth = Rational(1, 6)

>>> print(oneHalf)

1/2

>>> print(oneHalf + oneSixth)

2/3

>>> oneHalf == oneSixth

False

>>> oneHalf > oneSixth

True
```

Rational Numbers

- **The program contains the internal representation of a rational number and also its string representation**
- **The constructor gets the numerator and denominator as arguments and sets two instance variables to this information**
- **This method reduces the rational number to its lowest terms**
- **To reduce a rational number to its lowest terms the greatest common divisor (GCD) of the numerator and the denominator is calculated**
- **Then divide the numerator and the denominator by this GCD**
- **These tasks are assigned to two methods _reduce and _gcd**

Rational Number Arithmetic and Operator Overloading

- **The code x + y is actually shorthand for the code x.__add__(y)**

OPERATOR	METHOD NAME
+	__add__
-	__sub__
*	__mul__
/	__div__
%	__mod__

Built-in arithmetic operators and their corresponding methods

- **To overload an arithmetic operator just define a new method using the appropriate method name**
- **The code for each method applies a rule of rational number arithmetic**

Rational Number Arithmetic and Operator Overloading

TYPE OF OPERATION	RULE
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Mutiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$
Division	$n_1/d_1 / n_2/d_2 = n_1d_2 / d_1n_2$

Rules for rational number arithmetic

- **Each method builds and returns a new rational number that represents the result of the operation**
- **The parameter self is viewed as the left operand of the operator whereas the parameter other is viewed as the right operand**

Rational Number Arithmetic and Operator Overloading

- This is the code for the addition operation
- Operator overloading is another example of an abstraction mechanism

```
def __add__(self, other):
    """Returns the sum of the numbers."""
    #Self is the left operand and other is the right operand
    newNumer = self._numer * other._denom + \
        other._numer * self._denom
    newDenom = self._denom * other._denom
    return Rational(newNumer, newDenom)
```

Comparison Methods

- When the Python interpreter encounters one of the comparison operators it uses a corresponding method defined in the float or int class
- The simplest way to compare two rational numbers is to compare the product of the extremes and the product of the means
- The extremes are the first numerator and the second denominator whereas the means are the second numerator and the first denominator
- Thus the comparison of 1/6 and 2/3 translates to 1 * 3 < 2 * 6

Comparison Methods

OPERATOR	MEANING	METHOD
==	Equals	__eq__
!=	Not equals	__neq__
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__

```
def __lt__(self, other):
"""Compares two rational numbers self and other using < """
    extremes = self._numer * other._denom
    means = other._numer * self._denom
    return extremes < means
```

Equality and the __eq__ Method

- **The Python interpreter picks out equality by looking for an __eq__ method when it encounters the == and != operators**
- **The __eq__ method in a class can be used to support equality tests with any types of objects**
- **The method first tests the two operands for object identity using Python's is operator**
- **The is operator returns True if self and other refer to the exact same object**
- **If the two objects are distinct the method then uses Python's type function to determine whether or not they are of the same type**

Equality and the __eq__ Method

- **If they are not of the same type they cannot be equal**
- **Finally, if the two operands are of the same type the second one must be a rational number so it is safe to access the components of both operands to compare them for equality in the last alternative**

Equality and the __eq__ Method

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                   # Object identity?
        return True
    elif type(self) != type(other):     # Types match?
        return False
    else:
        return self._numer == other._numer and \
           self._denom == other._denom
```

Savings Accounts and Class Variables

- **A savings account allows owners to make deposits and withdrawals**
- **These accounts compute interest periodically**
- **A simplified version of a savings account includes an owner's name, PIN, and balance as attributes**
- **When the interest is computed a rate is applied to the balance**
- **The interest rate is same for all accounts so a class variable can be used to calculate the interest rate**
- **A class variable behaves like a constant in some situations a class variable can be modified**

Savings Accounts and Class Variables

SavingsAccount METHOD	WHAT IT DOES
`a = SavingsAccount(name, pin, balance = 0.0)`	Returns a new account with the given name, PIN, and balance.
`a.deposit(amount)`	Deposits the given amount to the account's balance.
`a.withdraw(amount)`	Withdraws the given amount from the account's balance.
`a.getBalance()`	Returns the account's balance.
`a.getName()`	Returns the account's name.
`a.getPin()`	Returns the account's PIN.
`a.computeInterest()`	Computes the account's interest and deposits it.
`__str__(a)`	Same as **`str(a)`**. Returns the string representation of the account.

Savings Accounts and Class Variables

```
class SavingsAccount(object):
"""This class represents a Savings account with the owner's
name, PIN, and balance."""
    RATE = 0.02
    def __init__(self, name, pin, balance = 0.0):
        self._name = name
        self._pin = pin
        self._balance = balance
    def __str__(self):
        result =  'Name:     ' + self._name + '\n'
        result += 'PIN:      ' + self._pin + '\n'
        result += 'Balance: ' + str(self._balance)
        return result
    def getBalance(self):
        return self._balance
```

Savings Accounts and Class Variables

```
def getName(self):
        return self._name
    def getPin(self):
        return self._pin
    def deposit(self, amount):
        """Deposits the given amount and returns the
        new balance."""
        self._balance += amount
        return self._balance
    def withdraw(self, amount):
        """Withdraws the given amount. Returns None if
successful, or an error message if unsuccessful."""
        if amount < 0:
            return 'Amount must be >= 0'
        elif self._balance < amount:
            return 'Insufficient funds'
```

Savings Accounts and Class Variables

```
else:
            self._balance -= amount
            return None
    def computeInterest(self):
        """Computes, deposits, and returns the interest."""
        interest = self._balance * SavingsAccount.RATE
        self.deposit(interest)
        return interest
```

Putting the Accounts into a Bank

- A simple bank allows a user to add new accounts, remove accounts, get existing accounts, and compute interest on all accounts
- A Bank class thus has these four basic operations (add, remove, get, and computeInterest) and a constructor
- This class also includes the usual str function for development and debugging

```
>>> from savingsAccount import Bank,SavingsAccount
>>> bank = Bank()
>>> bank.add(SavingsAccount("Wilma", "1001", 4000.00))
>>> bank.add(SavingsAccount("Fred", "1002", 1000.00))
```

Putting the Accounts into a Bank

```
>>> print(bank)
        Name:    Wilma
        PIN:     1001
        Balance: 4000.0
        Name:    Fred
        PIN:     1002
        Balance: 1000.0
>>> account = bank.get("1000")
>>> print(account)
None
>>> account = bank.get("1001")
>>> print(account)
        Name:    Wilma
        PIN:     1001
        Balance: 4000.0
```

Putting the Accounts into a Bank

```
>>> account.deposit(25.00)
        4025.0
>>> print(account)
        Name:    Wilma
        PIN:     1001
        Balance: 4025.0
>>> print(bank)
        Name:    Wilma
        PIN:     1001
        Balance: 4025.0
        Name:    Fred
        PIN:     1002
        Balance: 1000.0
```

Putting the Accounts into a Bank

Bank METHOD	WHAT IT DOES
`b = Bank()`	Returns a bank.
`b.add(account)`	Adds the given account to the bank.
`b.remove(pin)`	Removes the account with the given `pin` from the bank and returns the account. If the `pin` is not in the bank, returns `None`.
`b.get(pin)`	Returns the account associated with the `pin` if the PIN is in the bank. Otherwise, returns `None`.
`b.computeInterest()`	Computes the interest on each account, deposits it in that account, and returns the total interest.
`__str__(b)`	Same as `str(b)`. Returns a string representation of the bank (all the accounts).

The interface for the Bank class

Putting the Accounts into a Bank

```
class Bank(object):
    def __init__(self):
        self._accounts = {}
    def __str__(self):
        """Return the string rep of the entire bank."""
        return '\n'.join(map(str, self._accounts.values()))
    def add(self, account):
        """Inserts an account using its PIN as a key."""
        self._accounts[account.getPin()] = account
    def remove(self, pin):
        return self._accounts.pop(pin, None)
```

Putting the Accounts into a Bank

```
def get(self, pin):
        return self._accounts.get(pin, None)
    def computeInterest(self):
        """Computes interest for each account and
        returns the total."""
        total = 0.0
        for account in self._accounts.values():
            total += account.computeInterest()
        return total
```

Putting the Accounts into a Bank

```
def get(self, pin):
        return self._accounts.get(pin, None)
    def computeInterest(self):
"""Computes interest for each account and returns the total"""
        total = 0.0
        for account in self._accounts.values():
            total += account.computeInterest()
        return total
```

- Access and removal depend on an owner's PIN
- The use of the value None in the methods remove and get indicates to the user that the given PIN is not in the bank

Using pickle for Permanent Storage of Objects

- **An object can be converted to text for storage**
- **The mapping of complex objects to text and back again is a tedious process and cause maintenance headaches**
- **Python includes a module that allows the programmer to save and load objects using a process called pickling**
- **The term comes from the process of converting cucumbers to pickles for preservation in jars**
- **We can pickle an object before it is saved to a file and then unpickle it as it is loaded from a file into a program**
- **Python takes care of all of the conversion details automatically**

Using pickle for Permanent Storage of Objects

- First the pickle module should be imported
- Files are opened for input and output and closed in the usual manner except that the flags “rb” and “wb” are used instead of “r” and “w”, respectively
- To save an object the function pickle.dump is used
- The first argument is the object to be dumped or saved to a file and its second argument is the file object
- The pickle module can be used to save the accounts in a bank to a file
- Start by defining a Bank method named save
- The method includes an optional argument for the filename

Using pickle for Permanent Storage of Objects

- **The Bank object also has an instance variable for the filename**
- **For a new empty bank this variable's value is initially None**
- **Whenever the bank is saved to a file this variable becomes the current filename**
- **When the method's filename argument is not provided the method uses the bank's current filename if there is one**
- **This is similar to using the Save option in a File menu**
- **When the filename argument is provided it is used to save the bank to a different file**
- **This is similar to the Save As option in a File menu**

Using pickle for Permanent Storage of Objects

```
import pickle
def save(self, fileName = None):
"""Saves pickled accounts to a file.  The parameter allows the
user to change filenames."""
    if fileName != None:
        self._fileName = fileName
    elif self._fileName == None:
        return
    fileObj = open(self._fileName, 'wb')
    for account in self._accounts.values():
        pickle.dump(account, fileObj)
    fileObj.close()
```

Input of Objects and the try-except Statement

- **The pickled objects can be loaded into a program from a file using the function pickle.load**
- **If the end of the file has been reached this function raises an exception**
- **Python's try-except statement can be used here to catch the exception**
- **The try-except allows an exception to be caught and the program to recover**
- **The syntax of a simple try-except statement is**

```
try:
    <statements>
except <exception type>:
    <statements>
```

Input of Objects and the try-except Statement

- **When a statement is run the statements within the try clause are executed.**
- **If one of these statements raises an exception control is immediately transferred to the except clause**
- **If the type of exception raised matches the type in this clause its statements are executed**
- **Otherwise control is transferred to the caller of the try-except statement and further up the chain of calls until the exception is successfully handled or the program halts with an error message**

Input of Objects and the try-except Statement

- If the statements in the try clause raise no exceptions the except clause is skipped and control proceeds to the end of the try-except statement
- We can construct an input file loop that continues to load objects until the end of the file is encountered and when this happens an EOFError is raised
- The except clause then closes the file and breaks out of the loop
- Here is the code for a Bank method __init__ that can take some initial accounts from an input file

Input of Objects and the try-except Statement

```
def __init__(self, fileName = None):
"""Creates a new dictionary to hold the accounts. If a file name is
provided, loads the accounts from a file of pickled accounts."""
    self._accounts = {}
    self._fileName = fileName
    if fileName != None:
        fileObj = open(fileName, 'rb')
        while True:
            try:
                account = pickle.load(fileObj)
                self.add(account)
            except EOFError:
                fileObj.close()
                break
```

Encapsulation

- **In object-oriented programming encapsulation is a mechanism that enables the bundling of data and methods and restricts access to the data from outside the class**
- **Encapsulation is a technique for hiding a class internal operations and exposing only the information required for external interactions**
- **Encapsulation restricts access to a class's data to its methods and keeps the class's variables private**
- **Encapsulation acts as a protective shield that puts restrictions on accessing variables and methods directly and can prevent accidental or unauthorized modification of data**

Encapsulation

- **The three access levels in Python are**
  - **Public**
  - **Protected**
  - **Private**
- **Public variables and methods are those that can be accessed by any part of the program**
- **In Python variables and methods are public by default and they do not require any special keyword to be declared as public**

Encapsulation

```
class Rectangle:

    length = 5

    breadth = 10

    def area(self):

        return self.length * self.breadth

r = Rectangle()

print(r.length)

print(r.breadth)

print(r.area())
```

Encapsulation

Output

5

10

50

- **In the example the length, breadth variables and the area method are public and can be accessed by any part of the program**

Encapsulation

- **Protected variables and methods are those that can be accessed within the class and its subclasses**
- **In Python protected variables and methods are declared by prefixing them with a single underscore (_)**
- **In the example the variables _length, _breadth and the method _area are protected and can be accessed within the Rectangle class and its subclasses r**

Encapsulation

```
class Rectangle:

    _length = 5

    _breadth = 10

    def _area(self):

        return self._length * self._breadth

r = Rectangle()

print(r._length)

print(r._breadth)

print(r._area())
```

Encapsulation

Output

5

10

50

Encapsulation

- **Private variables and methods are those that can only be accessed within the class**
- **In Python private variables and methods are declared by prefixing them with double underscores (__)**
- **In the example, the variables __length, __breadth and the method __area are private and can only be accessed within the Rectangle class**

Encapsulation

```
class Rectangle:

    __length = 5

    __breadth = 10

    def __area(self):

        return self.__length * self.__breadth

r = Rectangle()

print(r.__length)

print(r.__breadth)

print(r.__area())
```

Encapsulation

Output

Traceback (most recent call last): File "C:\Users\HP\Desktop\Python\Module4\programs\Encapsulation3.py", line 10, in <module> print(r.__length)

AttributeError: 'Rectangle' object has no attribute '__length'

- **Private variables cannot be accessed outside the class and private methods can only be called from within the class**
- **When private variables and class are accessed outside the class they give error telling the object does not have an attribute by that name**

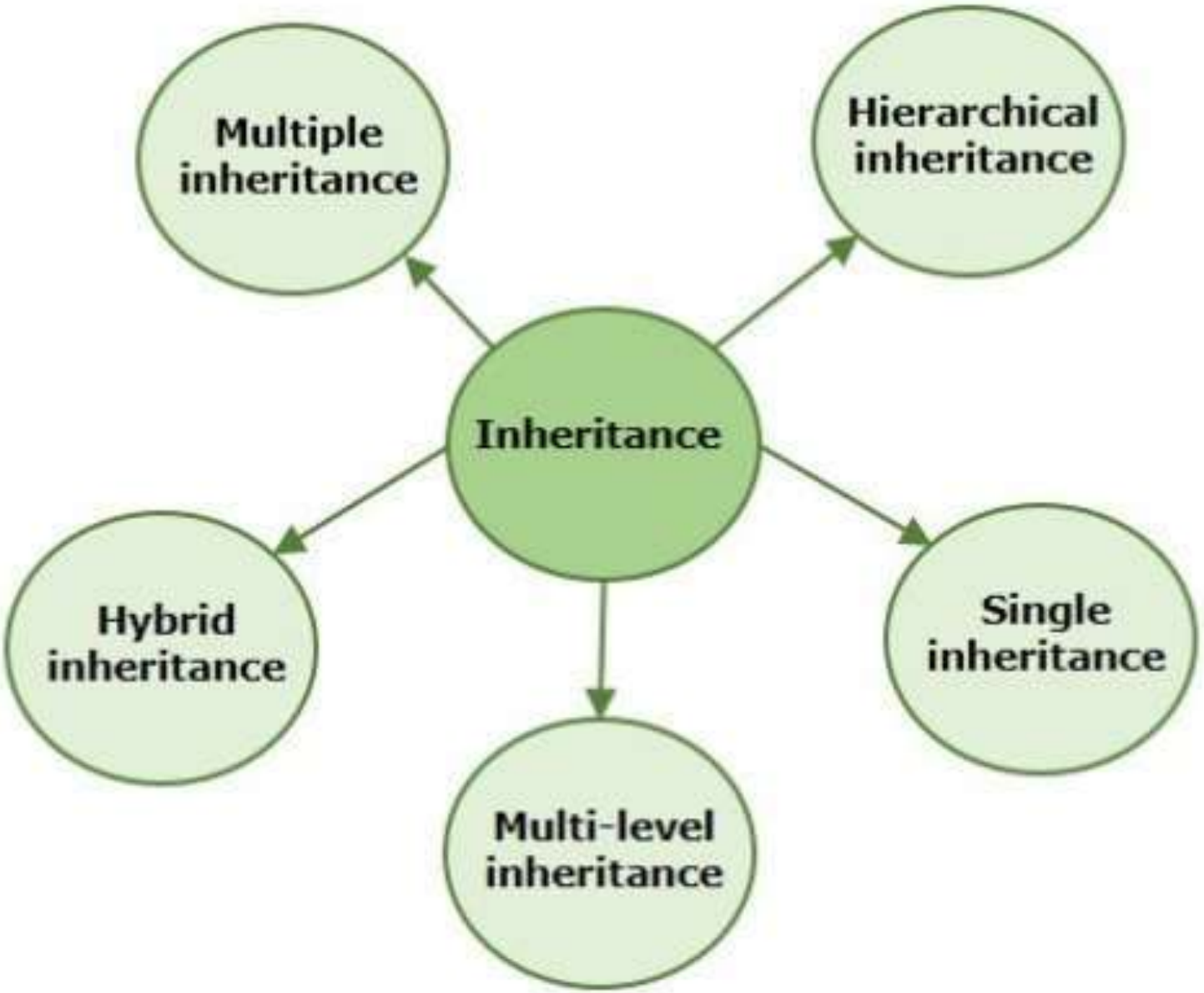
Inheritance

- **Inheritance is used to inherit the properties and behaviours of one class to another**
- **The class that inherits another class is called a *child class***
- **The class that gets inherited is called a *base class* or *parent class***
- **Inheritance allows capabilities of existing class to be reused and if required extended to design a new class**
- **Inheritance can be used to avoid code repetition**
- **The advantage of using inheritance include reusability, saving time and effort, data hiding, extensibility, easy understandability, reliability and maintainability**

Inheritance

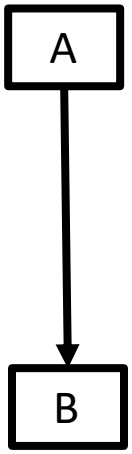
- **In Python inheritance is divided into five different categories**
  - **Single Inheritance**
  - **Multiple Inheritance**
  - **Multilevel Inheritance**
  - **Hierarchical Inheritance**
  - **Hybrid Inheritance**

Inheritance



Single Inheritance

- **Single Inheritance is the simplest form of inheritance where a child class inherits attributes and methods from only one parent class**
- **It enables a child class to adopt or inherit properties and methods from one parent class only**



Single Inheritance

```
#types of inheritance in python - Single Inheritance

class Animal:

    def __init__(self, name):

        self.name = name

    def speak(self):

        print(f"{self.name} makes a noise")

class Dog(Animal):

    def __init__(self, name, breed):

        super().__init__(name)

        self.breed = breed
```

Single Inheritance

```
def bark(self):
        print(f"{self.name}, the {self.breed}, barks")
# Creating an instance of Dog
dog = Dog("Buddy", "Golden Retriever")
dog.speak()  # Inherited method
dog.bark()   # Own method
```

Single Inheritance

Output

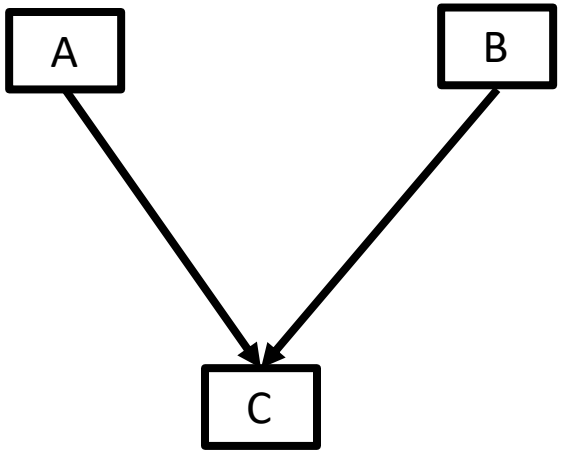
Buddy makes a noise

Buddy, the Golden Retriever, barks

- **Single inheritance is quite beneficial when there is a clean and uncomplicated line of succession**
- **eg. a Dog class can extend and inherit from the Animal class**
- **Here Code reusability is promoted and repetition is distaste**

Multiple Inheritance

- **Multiple inheritances refer to a situation whereby one class is derived from more than one parent class**
- **Here the child class getting the functionality from the parent classes but also adds some complexity**
- **The child can override methods inherited from any parent**



Multiple Inheritance

```
#types of inheritance in python - Multiple Inheritance

class Flyer:

    def fly(self):

        print("Flying high")

class Swimmer:

    def swim(self):

        print("Swimming in the water")

class Duck(Flyer, Swimmer):

    def quack(self):

        print("Quacking")
```

Multiple Inheritance

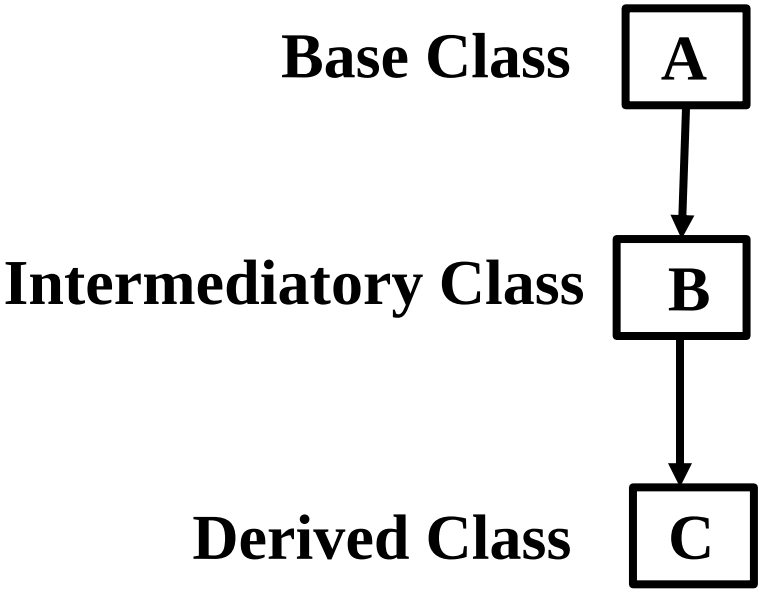
```
# Creating an instance of Duck

duck = Duck()

duck.fly()   # Inherited from Flyer

duck.swim()  # Inherited from Swimmer

duck.quack() # Own method
```

Output

```
Flying high

Swimming in the water

Quacking
```

Multilevel Inheritance

- **In Multilevel Inheritance a class is inherited from a class which in turn is inherited from another class.**
- **Multilevel Inheritance leads to an inheritance pattern that gives the way to making new more complicated hierarchies**
- **It is like a grandparent-parent-child relationship**



Multilevel Inheritance

```
#types of inheritance in python - Multilevel
Inheritance
class Animal:
    def eat(self):
        print("Eating")
class Mammal(Animal):
    def walk(self):
        print("Walking")
class Dog(Mammal):
    def bark(self):
        print("Barking")
```

Multilevel Inheritance

```
# Creating an instance of Dog

dog = Dog()

dog.eat()   # Inherited from Animal

dog.walk()  # Inherited from Mammal

dog.bark()  # Own method
```

Output

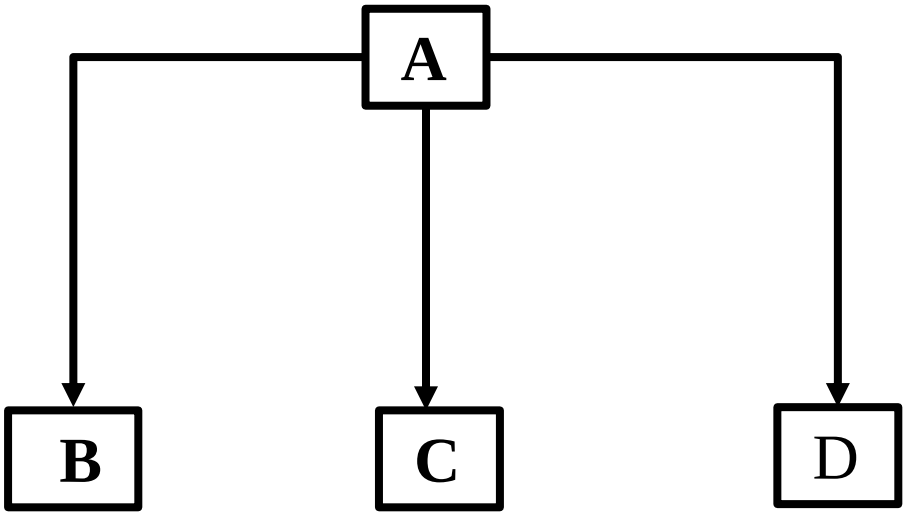
```
Eating

Walking

Barking
```

Multilevel Inheritance

- **Multilevel inheritance will be helpful when using classes for creating complex relationships**
- **Multilevel inheritance comes with a number of problems concerning maintainability**
- **The changes in higher levels of the hierarchy can affect derived classes**

Hierarchical Inheritance

- **In Hierarchical Inheritance concept one parent class is used to create multiple classes and all the classes have the same parent**
- **Such inheritance is most useful when creating classes that should contain common sets of methods and properties**
- **Multiple classes are derived from a single base class**



Hierarchical Inheritance

```
#types of inheritance in python - Hierarchical
Inheritance
class Animal:
    def speak(self):
        print("Animal speaks")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
class Cat(Animal):
    def meow(self):
        print("Cat meows")
```

Hierarchical Inheritance

```
# Creating instances of Dog and Cat

dog = Dog()

cat = Cat()

dog.speak()  # Inherited from Animal

dog.bark()   # Own method

cat.speak()  # Inherited from Animal

cat.meow()   # Own method
```

Hierarchical Inheritance

Output

Animal speaks

Dog barks

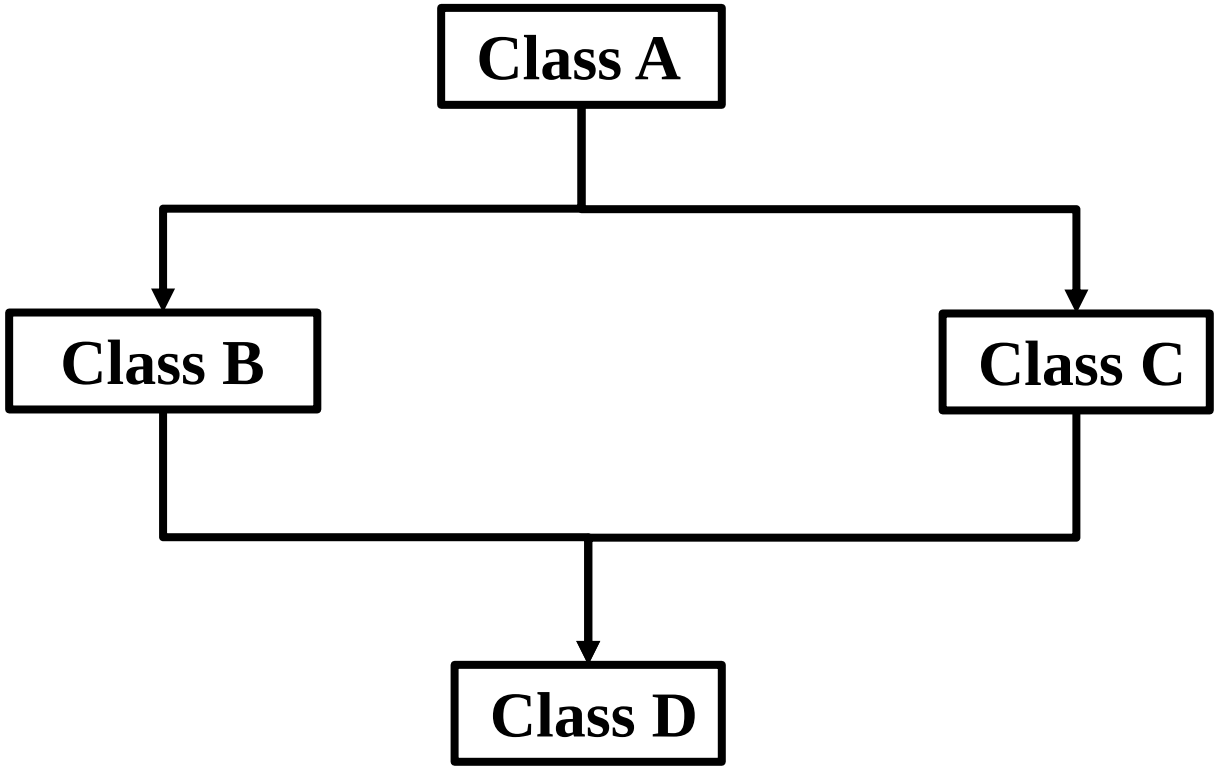
Animal speaks

Cat meows

- **Hierarchical Inheritance is used in a way that would allow testing of general behaviors or characteristics of a parent class and then have those behaviors checked in several child classes**

Hybrid Inheritance

- **Hybrid inheritance is also known as multiple type inheritance whereby an object can have two or more types of inheritance**
- **It is used commonly in high-level designs and is more flexible than the first one**



Hybrid Inheritance

```
#types of inheritance in python - Hybrid

Inheritance

class Vehicle:

    def move(self):

        print("Vehicle is moving")

class Car(Vehicle):

    def drive(self):

        print("Car is driving")

class Boat(Vehicle):

    def sail(self):

        print("Boat is sailing")
```

Hybrid Inheritance

```
class AmphibiousVehicle(Car, Boat):
    def drisail(self):
        print("Sail in water and Drive in land")
# Creating an instance of AmphibiousVehicle
amphibious_vehicle = AmphibiousVehicle()
amphibious_vehicle.move()  # Inherited from Vehicle
amphibious_vehicle.drive() # Inherited from Car
amphibious_vehicle.sail()  # Inherited from Boat
amphibious_vehicle.drisail()
```

Hybrid Inheritance

Output

Vehicle is moving

Car is driving

Boat is sailing

Sail in water and Drive in land

Hybrid Inheritance

- **Hybrid inheritance is a little complicated since there will be multiple inheritance channels**
- **Python's Method resolution order (MRO) system guarantees the usage of the correct method or attribute when the path linkage is multiple**
- **MRO should be used sparingly when developing a system since it offers flexibility in the way a complicated system is developed**

Method resolution order (MRO)

- **MRO stands for Method Resolution Order**
- **It is the order in which Python looks for a method in a hierarchy of classes**
- **MRO follows a specific sequence to determine which method to invoke when it encounters multiple classes with the same method name**
- **In the next example the methods that are invoked is from class Child but not from class Father and this is due to Method Resolution Order(MRO).**
- **The order that follows in the next code is class Child – > class Father**

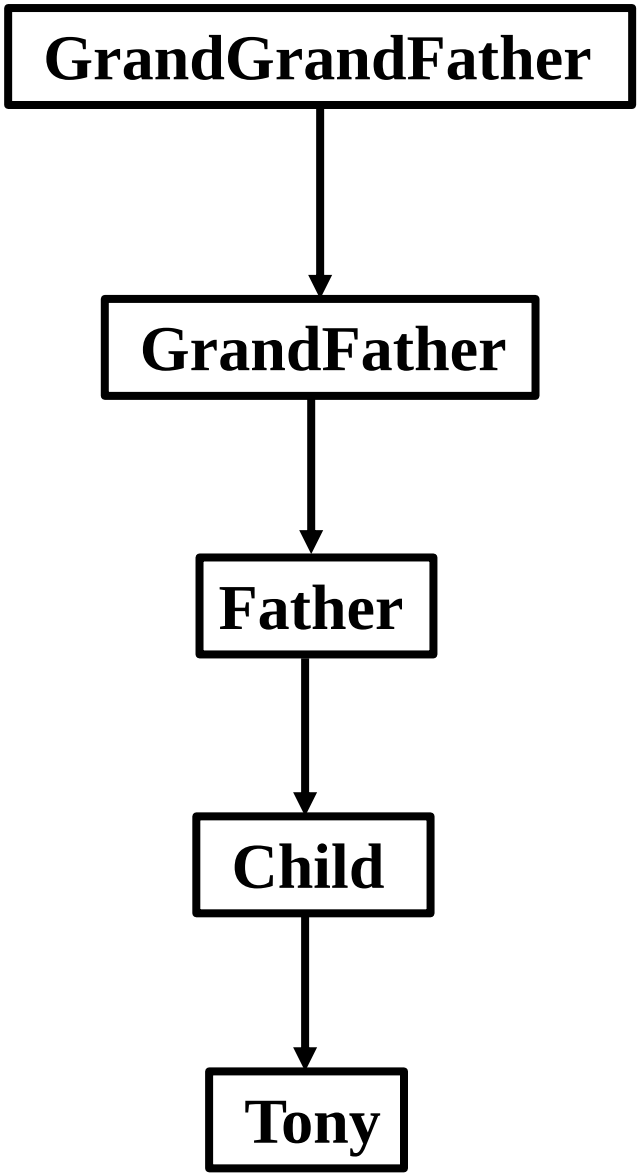
Method resolution order (MRO)

```
class Father:
    def hobby(self):
        print("Traveling")
class Child(Father):
    def hobby(self):
        print("Cycling")
tony = Child()
tony.hobby()
```

Output

```
Cycling
```

Method resolution order (MRO)



Method resolution order (MRO)

```
class GrandGrandFather:
    def hobby(self):
        print("Fishing")
class GrandFather(GrandGrandFather):
    def hobby(self):
        print("Gardening")
class Father(GrandFather):
    def hobby(self):
        print("Traveling")
class Child(Father):
    def hobby(self):
        print("Cycling")
```

Method resolution order (MRO)

```
class Tony(Child):
    pass
t = Tony()
t.hobby()
```

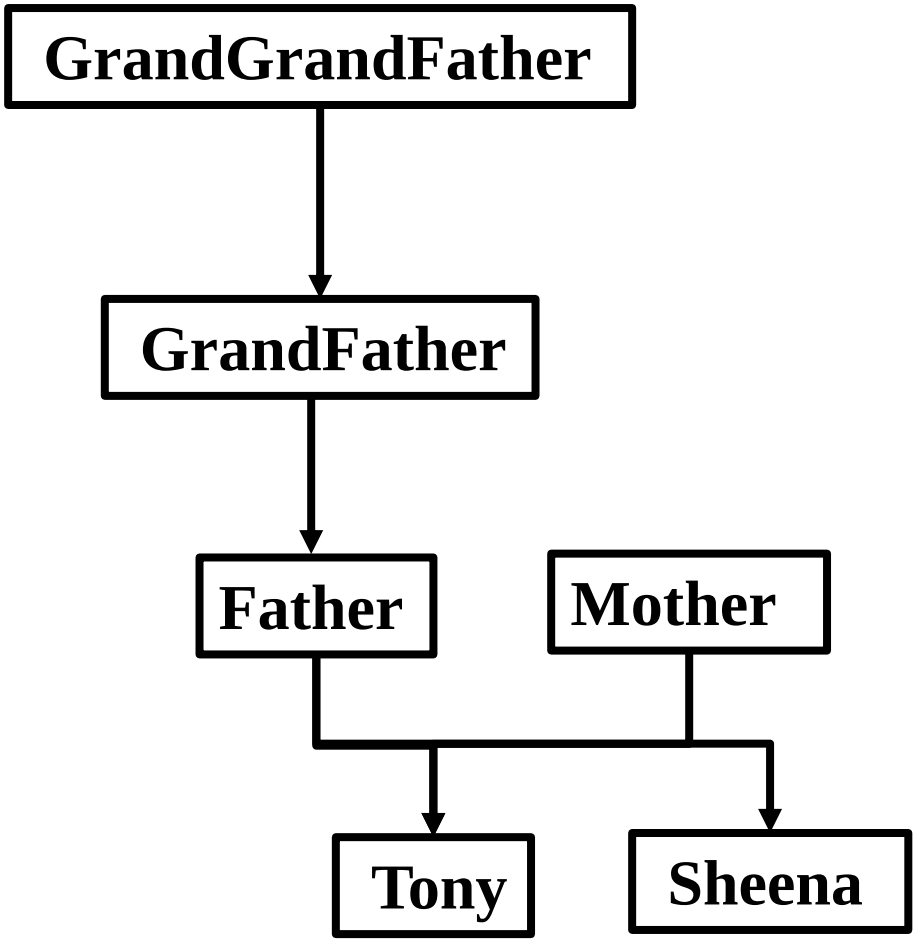
Output

```
Cycling
```

Method resolution order (MRO)

- **In multiple inheritances the methods are executed based on the order specified while inheriting the classes**
- **For python multiple inheritance method resolution order plays a very crucial role**

Method resolution order (MRO)



Method resolution order (MRO)

```
class GrandGrandFather:
    def hobby(self):
        print("Fishing")
class GrandFather(GrandGrandFather):
    def hobby(self):
        print("Gardening")
class Father(GrandFather):
    def hobby(self):
        print("Traveling")
class Mother:
    def hobby(self):
        print("Singing")
```

Method resolution order (MRO)

```
class Tony(Father, Mother):
    pass
class Sheena(Mother, Father):
    pass
t = Tony()
t.hobby()
s = Sheena()
s.hobby()
print(Tony.mro())
print(Sheena.mro())
```

Method resolution order (MRO)

Output

Traveling

Singing

[<class '__main__.Tony'>, <class '__main__.Father'>, <class '__main__.GrandFather'>, <class '__main__.GrandGrand Father'>, <class '__main__.Mother'>, <class 'object'>]

[<class '__main__.Sheena'>, <class '__main__.Mother'>, <class '__main__.Father'>, <class '__main__.GrandFather'>, <class '__main__.GrandGrandFather'>, <class 'object'>]

Method resolution order (MRO)

- **Method resolution order is determined by the C3 linearization algorithm**
- **This algorithm ensures a consistent and predictable order for method lookup in multiple inheritance scenarios**
- **Rules of C3 Linearization**
- **The C3 Linearization algorithm enforces two key rules:**

1. **Children before Parents: If class A inherits from class B, class A must appear before class B in the MRO**
2. **Depth-First, Left-to-Right: The algorithm prioritizes the order of inheritance defined in the class definition, with a left-to-right depth-first traversal**

Method resolution order (MRO)

- **In the example algorithm first looks into the instance class for the invoked method**
- **If not present then it looks into the first parent if that too is not present then parent of the parent is looked into**
- **This continues till the end of the depth of class and finally till the end of inherited classes**
- **To get the method resolution order of a class we can use mro() method**
- **By using these methods we can display the order in which methods are resolved**

Polymorphism

- **Polymorphism in Python**
- **Types of Polymorphism**
- **Duck Typing**
- **Overloading**
    - **Operator Overloading**
    - **Method Overloading**
    - **Constructor Overloading**
- **Overriding**
    - **Method Overriding**

Polymorphism

- **Polymorphism refers to the ability of objects to take on multiple forms or behave differently based on the context**
- **The word polymorphism means having many forms and it comes from the Greek words poly (many) and morph (forms) this means one entity can take many forms**
- **Polymorphism allows the same method or object to behave differently based on the context specially on the project's actual runtime**
- **There are some functions in Python which are compatible to run with multiple data types**
- **The len() function can run with many data types in Python**

Polymorphism

- **Many data types such as string, list and dictionary can work with the len() function**

```
print(len("Programiz"))
print(len(["Python", "Java", "C","R"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Output

```
9
4
2
```

Polymorphism

- **Polymorphism in Python can be implemented in two ways**
  1. **Compile-time (static)**
  2. **Run-time (dynamic)**
- **Compile-time polymorphism also known as static polymorphism is achieved through**
  1. **Method overloading**
  2. **Operator overloading**
- **Run-time polymorphism is implemented through**
  1. **Method overriding**

Duck Typing

- **Duck typing is a concept in programming that gives emphasizes more on the object's behavior over its type or class**
- **When duck typing is used it checks whether an object has certain methods or attributes rather than checking for the exact class**
- **The name comes from the saying**

  " If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck "
- **Duck typing is a form of runtime polymorphism**
- **If an object can perform certain actions or has certain attributes it's treated as if it belongs to a certain type or class even if it doesn't explicitly inherit from that type or class**

Duck Typing

```
class Dog:
    def speak(self):
        return "Woof!"
class Cat:
    def speak(self):
        return "Meow!"
class Duck:
    def speak(self):
        return "Quack!"
def animal_sound(animal):
    return animal.speak()
```

Duck Typing

```
dog = Dog()
cat = Cat()
duck = Duck()
print(animal_sound(dog))  # Output: Woof!
print(animal_sound(cat))  # Output: Meow!
print(animal_sound(duck)) # Output: Quack!
```

Output

```
Woof!
Meow!
Quack!
```

Duck Typing

- **The animal_sound function takes an argument that is expected to have a speak method**
- **Duck typing allows to pass objects of different classes (Dog, Cat, and Duck) to the function as long as they have a compatible method (speak).**
- **The function doesn't care about the actual type of the object it only relies on the presence of the required method**

Duck Typing

```
class Sparrow:
    def fly(self):
        print("Sparrow flying")
class Airplane:
    def fly(self):
        print("Airplane flying")
class Whale:
    def swim(self):
        print("Whale swimming")
def lift_off(entity):
    entity.fly()
```

Duck Typing

sparrow = Sparrow()

airplane = Airplane()

whale = Whale()

lift_off(sparrow) # prints `Sparrow flying`

lift_off(airplane) # prints `Airplane flying`

lift_off(whale) # Throws the error `'Whale' object has no attribute 'fly'`

- **The ‘whale' object of Whale() class do not have any attribute called 'fly’**
- **The function lift_off expects all the objects to have a fly function but Whale class do not have a fly function**

Duck Typing

Sparrow flying

Airplane flying

Traceback (most recent call last): File "C:\Users\HP\Desktop\Python\Module4\programs\duckTyping2.py", line 17, in <module>

lift_off(whale) # Throws the error `'Whale' object has no attribute 'fly'`

File"C:\Users\HP\Desktop\Python\Module4\programs\duckTyping2.py", line 11, in lift_off

entity.fly()

AttributeError: 'Whale' object has no attribute 'fly'

Operator Overloading

- **Operator Overloading is the ability of an operator to behave in different ways depending on the parameters that are passed to the operator or the operands that the operator acts on**
- **Operator overloading refers to providing extended meaning to operators beyond the intended or predefined operational meaning**
- **The + operator is used to add two integers, join two strings, and merge two lists**
- **A user can overload existing operators but can not create a new operator**

Operator Overloading

- **If a person uses the + operator it will invoke the function __add__ automatically in the command line where the + operator is defined**
- **The following program add two Coordinates(points) using operator overloading**
- **Comparison operator overloading goes beyond binary operators and extends to other Python operators, such as <, >, <=, >=, ==, and !=**

Operator Overloading

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    def __str__(self):
        return f"({self.x}, {self.y})"
p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2
print(p3)  # Output: (6, 8)
```

Operator Overloading

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width
        self.area = length * width
    def __lt__(self, other):
        return self.area < other.area
    def __le__(self, other):
        return self.area <= other.area
    def __gt__(self, other):
        return self.area > other.area
```

Operator Overloading

```
def __ge__(self, other):
        return self.area >= other.area
    def __eq__(self, other):
        return self.area == other.area
    def __ne__(self, other):
        return self.area != other.area
    def __str__(self):
        return f"Rectangle({self.length} x {self.width})"
# Creating two rectangle objects
rect1 = Rectangle(4, 5)
rect2 = Rectangle(3, 7)
```

Operator Overloading

 # Comparing the rectangles using overloaded comparison operators

```
print(rect1 > rect2)   # Output: False
print(rect1 < rect2)   # Output: True
print(rect1 >= rect2)  # Output: False
print(rect1 <= rect2)  # Output: True
print(rect1 == rect2)  # Output: False
print(rect1 != rect2)  # Output: True
```

Operator Overloading

Output

False

True

False

True

False

True

Method Overloading

- **Method Overloading is a property in Object Oriented programming where two or more methods have the same name but differ in the number and type of parameters**
- **To overload method we must change the number of parameters or the type of parameters or both**
- **Python does not directly support method overloading like Java**
- **In python method overloading like functionality can be achieved by using techniques like default arguments, *args, and **kwarg**

Method Overloading

```
def add(a, b, c=0, d=0, e=0):
    return a + b + c + d + e
print(add(1, 2, 3, 4, 5))
print(add(1, 2.5, 3.7, 4))
print(add(1, 2, 3))
print(add(1, 2.8))
```

Output

```
15
11.2
6
3.8
```

Method Overloading

```
def multi(a, b, c=1, d=1, e=1):
    return a * b * c * d * e
print(multi(2, 3, 4, 5, 6))
print(multi(2.5, 3.7, 4, 5))
print(multi(2, 3.2, 4))
print(multi(2 ,3))
```

Output

```
720
185.0
25.6
6
```

*args

- In python *args allows us to pass a variable number of arguments to a function
- In the function we should use an asterisk (*) before the parameter name to pass a variable number of arguments
- It is used to pass a variable-length argument list
- *args allows to pass a variable number of non keyword arguments to function
- The arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk
- *args are used with lists

*args

```
def adder(*num):
    sum = 0
    for n in num:
        sum = sum + n
    print("Sum:",sum)
adder(3,5)
adder(4,5,6,7)
adder(1,2,3,5,6)
adder(1,2,3,4,5,6,7,8,9,10)
```

*args

Output

Sum: 8

Sum: 22

Sum: 17

Sum: 55

**kwargs

- **In python kwargs stands for keyword arguments**
- **kwargs** allows to pass a variable number of keyword arguments to a Python function**
- **The arguments are passed as a dictionary and these arguments make a dictionary inside the function with name same as the parameter excluding ****
- **A keyword argument is where you provide a name to the variable as you pass it into the function**
- **It collects all the additional keyword arguments passed to the function and stores them in a dictionary**

**kwargs

```
def var_dict(**valus):
    print(valus)


var_dict(a=5, b=6, c=7)
```

Output

```
{'a': 5, 'b': 6, 'c': 7}
```

**kwargs

```
def intro(**data):
    print("\nData type of argument:",type(data))
    for key, value in data.items():
        print("{} is {}".format(key,value))
intro(Firstname="Lakshmi", Lastname="Ramkumar", Age=22,
Phone=1234567890)
intro(Firstname="John", Lastname="Thomas",
Email="johntom@nomail.com", Country="India", Age=25,
Phone=9876543210)
```

**kwargs

Output

Data type of argument: <class 'dict'>

Firstname is Lakshmi

Lastname is Ramkumar

Age is 22

Phone is 1234567890

Data type of argument: <class 'dict'>

Firstname is John

Lastname is Thomas

Email is johntom@nomail.com

Country is India

Age is 25

Phone is 9876543210

Constructor Overloading

- **Python does not directly support constructor overloading like Java**
- **But constructor overloading can be simulated by using default arguments, variable-length arguments (*args, **kwargs), or conditional logic within the __init__ method**

Constructor Overloading

```
class Rectangle:
    def __init__(self, length=1, width=1):
        self.length = length
        self.width = width
    def calculate_area(self):
        return self.length * self.width
rect1 = Rectangle()
rect2 = Rectangle(5)
rect3 = Rectangle(5, 10)
```

Constructor Overloading

```
area1 = rect1.calculate_area()

print(area1)

area2 = rect2.calculate_area()

print(area2)

area3 = rect3.calculate_area()

print(area3)
```

Output

```
1

5

50
```

Constructor Overloading

```
class Vector:
    def __init__(self, *args):
        if len(args) == 0:
            self.coordinates = (0, 0, 0)
        else:
            self.coordinates = args
# Creating vectors with different number of arguments
vector1 = Vector()
vector2 = Vector(1, 2)
vector3 = Vector(1, 2, 3)
```

Constructor Overloading

```
print(vector1.coordinates)

print(vector2.coordinates)

print(vector3.coordinates)
```

Output

```
(0, 0, 0)

(1, 2)

(1, 2, 3)
```

Method Overriding

- **In object-oriented programming inheritance allows a child class to inherit attributes and behaviors from a parent class**
- **One powerful technique is method overriding which enables a child class to provide its own implementation for a method inherited from the parent class**
- **The child class will inherit all the methods from the parent class**
- **The child class can override specific inherited methods by redefining them using the same method signature**

Method Overriding

- **When the method is called on an instance of the child class its overriding implementation will be executed instead of the parent's version**
- **In the following program we have the parent class Animal which has a method speak**
- **Dog class is a child class of parent class Animal so the speak method is inherited to Dog class but the Dog class does not override the speak method**
- **Cat class is a child class of parent class Animal so the speak method is inherited to Cat class and the Cat class overrides the speak method**

Method Overriding

```
class Animal:
  def speak(self):
    print("The animal makes a sound")
class Dog(Animal):
        pass
dog = Dog()
dog.speak()
class Cat(Animal):
  def speak(self):
    print("The cat Meows!")
cat = Cat()
cat.speak()
```

Method Overriding

Output

The animal makes a sound

The cat Meows!

Abstract Classes

- **An abstract class cannot be instantiated on its own and is designed to be a blueprint or template for other classes**
- **Abstract classes allow us to define methods that must be implemented by subclasses while allowing the subclasses to provide specific implementations**
- **Abstract class defines methods that must be included in any class that inherits from it but it does not provide the actual code for those methods**
- **An abstract class cannot be directly used to create an object**
- **Abstract classes can have both concrete and abstract methods**

Abstract Classes

- **The abc module in Python offers built-in support for abstract classes**
- **The abc module stands for "Abstract Base Classes" gives programmers the necessary tools to create abstract classes, including the abstract method decorator and the ABC class**
- **By defining abstract methods and requiring their implementation in subclasses, this module helps to maintain uniformity and conformity to a predetermined interface**
- **Any class in python must inherit from ABC to define an abstract class**
- **An abstract method can be defined inside an abstract class by using the abstractmethod decorator "@abstractmethod"**

Abstract Classes

```
from abc import ABC, abstractmethod

class Shape(ABC):
    def __init__(self, color):
        self.color = color
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
    def display_color(self):
        return f"This color is {self.color}"
```

Abstract Classes

```
class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius
    def perimeter(self):
        return 2 * 3.14 * self.radius
class Rectangle(Shape):
    def __init__(self, color, length, width):
        super().__init__(color)
        self.length = length
```

Abstract Classes

```
self.width = width

    def area(self):

        return self.length * self.width

    def perimeter(self):

        return 2 * (self.length + self.width)

# Creating instances of concrete classes

#s = Shape()

circle = Circle("red", 5)

rectangle = Rectangle("blue", 10, 5)
```

Abstract Classes

```
# Calling methods
print("Area of circle:", circle.area())
print("Perimeter of circle:", circle.perimeter())
print(circle.display_color())

print("Area of rectangle:", rectangle.area())
print("Perimeter of rectangle:", rectangle.perimeter())
print(rectangle.display_color())
```

Abstract Classes

Output

Area of circle: 78.5

Perimeter of circle: 31.400000000000002

This color is red

Area of rectangle: 50

Perimeter of rectangle: 30

This color is blue

Abstract Classes

- **If the abstract class has abstract methods the children of the abstract class should implement the abstract methods**
- **Shape is an abstract class with abstract methods area and perimeter and a concrete method display_color**
- **Circle and Rectangle are concrete classes that inherit from Shape and implement the abstract methods**
- **An attempt to instantiate the abstract class Shape directly will raise a TypeError**

Interface

- **An interface is a contract that defines a set of methods a class must implement**
- **It specifies what a class should do without dictating how it is done**
- **Python does not have an explicit interface keyword**
- **Interfaces are not necessary in Python because Python has proper multiple inheritance**
- **Instead interfaces are typically implemented using abstract base classes (ABCs) from the abc module**

Interface

- **Interfaces specifies a set of methods that a class must implement in order to be considered compatible with the interface**
- **Interfaces are created by using abstract classes which have only abstract methods**
- **Abstract methods are those methods without implementation or which are without the body**
- **So the interface just defines the abstract method without implementation**
- **The implementation of these abstract methods is defined by classes that implement an interface**

Interface

- **Interface can contain:**
    - **Constructors**
    - **Variables**
    - **Abstract methods**
    - **Sub class**
- **In the following example Animal is an abstract base class acting as an interface**
- **The speak and move methods are abstract methods which are declared in the interface**
- **Dog and Cat classes implement this interface by providing concrete implementations for speak and move methods**

Interface

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
    @abstractmethod
    def move(self):
        pass
class Dog(Animal):
    def speak(self):
        print("Woof!")
```

Interface

```
def move(self):
        print("Runs on four legs")
class Bird(Animal):
    def speak(self):
        print("Chirp!")
    def move(self):
        print("Flies with wings")
# Demonstrating the interface
dog = Dog()
dog.speak()
dog.move()
```

Interface

```
bird = Bird()

bird.speak()

bird.move()
```

Output

```
Woof!

Runs on four legs

Chirp!

Flies with wings
```

Difference between Abstract Class and Interface in Python

Feature	Abstract Class	Interface
Methods	Can have both abstract and concrete methods	Can only have abstract methods (method signatures without implementation)
Implementation	Can provide default implementations for methods	Cannot contain any implementation details
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Inheritance	A class can inherit from only one abstract class	A class can implement multiple interfaces
Purpose	Defines a common base for related classes, may share code	Defines a contract for what a class can do, promoting flexibility
Variables	Can have instance variables	Usually does not have instance variables (can have static constants)

Exception handling

- **An unwanted or unexpected event which disturbs the normal flow of the program is called exception**
- **Exception handling in Python is a mechanism to manage runtime errors preventing program crashes**
- **Exception handling allows to respond to an error instead of crashing the running program**
- **In order to get program executed normally, handle those exceptions on high priority**
- **In python for every exception type a corresponding class is available and every exception is an object to its corresponding class**

Exception handling

- **Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors**
- **For instance they occur when we**
  - **try to open a file(for reading) that does not exist (FileNotFoundError)**
  - **try to divide a number by zero (ZeroDivisionError)**
  - **try to import a module that does not exist (ImportError) and so on**
- **Whenever these types of runtime errors occur Python creates an exception object and will check for handling code**

Exception handling

- **Here is the syntax of a simple try...except block in Python**

```
try:
    # Code that might raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...
```

- **The try block encloses the code that might potentially raise an exception**
- **If an exception occurs within this block Python will immediately jump to the except block that matches the type of exception raised**

Exception handling

- **The except block then handles the exception preventing the program from crashing**
- **If no exception is raised in the try block the except block is skipped**
- **The single statement print(x) when run gives an Error**
- **The Error states that**

  NameError: name 'x' is not defined
- **This Error can be caught by the try – except block as shown in the following slide**

Exception handling

```
try:
    print(x)
except NameError:
     print("Variable x is not defined")
```

Output

Variable x is not defined

Exception handling

- **Here is the syntax of a simple try...except block with finally in Python**

```
try:
    # Code that might raise an exception
    ...
except ExceptionType:
    # Code to handle the exception if it occurs
    ...
finally:
    # Code that will be executed regardless of whether an exception occurred or not
    ...
```

Exception handling

```
def access_list(my_list, index):
    try:
        value = my_list[index]
        print(f"Value at index {index}: {value}")
    except IndexError:
        print(f"Error: Index {index} is out of range.")
    finally:
        print("List access attempt complete.")
my_list = [10, 20, 30]
access_list(my_list, 1)
access_list(my_list, 5)
```

Exception handling

Output

Value at index 1: 20

List access attempt complete.

Error: Index 5 is out of range.

List access attempt complete.

Handle Multiple Exception

- **In handling multiple exceptions a program can respond to different exceptions without terminating it**
- **In Python try-except blocks can be used to catch and respond to one or multiple exceptions**
- **In cases where a process raises more than one possible exception they can all be handled using a single except clause**
- **Using a single try except block multiple exceptions can be caught**

Handle Multiple Exception

```
value = int(input("Enter a number : "))
try:
    # Code that might raise multiple exceptions
    result = 10 / value
    print(result)
    my_list = [1, 2, 3]
    print(my_list[value])
except (ZeroDivisionError, ValueError, IndexError) as e:
    print(f"An error occurred: {e}")
```

Handle Multiple Exception

Output

Enter a number : 5

2.0

An error occurred: list index out of range

Enter a number : 0

An error occurred: division by zero

Handle Multiple Exception

- **Using separate except blocks to catch multiple exceptions allows to handle different exception types in different ways**
- **Having separate except blocks is ideal to perform specific actions for each type of exception**

Handle Multiple Exception

- **Here is the syntax of multiple try except block in Python**

```
try:
    # Code that might raise an exception
except ExceptionType1:
    # Code to handle the exception1 if it occurs
except ExceptionType2:
    # Code to handle the exception2 if it occurs
except ExceptionType3:
    # Code to handle the exception2 if it occurs
finally:
  # Code will be executed regardless of exception occurrence
```

Handle Multiple Exception

```
try:
    x = int(input("Enter the first Number:"))
    y = int(input("Enter the second Number:"))
    print(x/y)
except ZeroDivisionError:
    print("ZeroDivisionError: Cannot divide with Zero")
except ValueError:
    print("ValueError: Please enter a Valid integer")
finally:
    print("The Division is Over")
```

Handle Multiple Exception

Output

Enter the first Number:25

Enter the second Number:0

ZeroDivisionError: Cannot divide with Zero

The Division is Over

Enter the first Number:45

Enter the second Number:a

ValueError: Please enter a Valid integer

The Division is Over