
Fundamentals of Programming

Pradeep Aryal M.Sc.

Contents

- Introduction to Computer
- Algorithm and Flow Chart
- Introduction to programming
- Software Development Methodology
- Datatypes and Variables
- Operators in a programming Language
- Conditions and Loops using Algorithm and Flowchart

Introduction to Computer

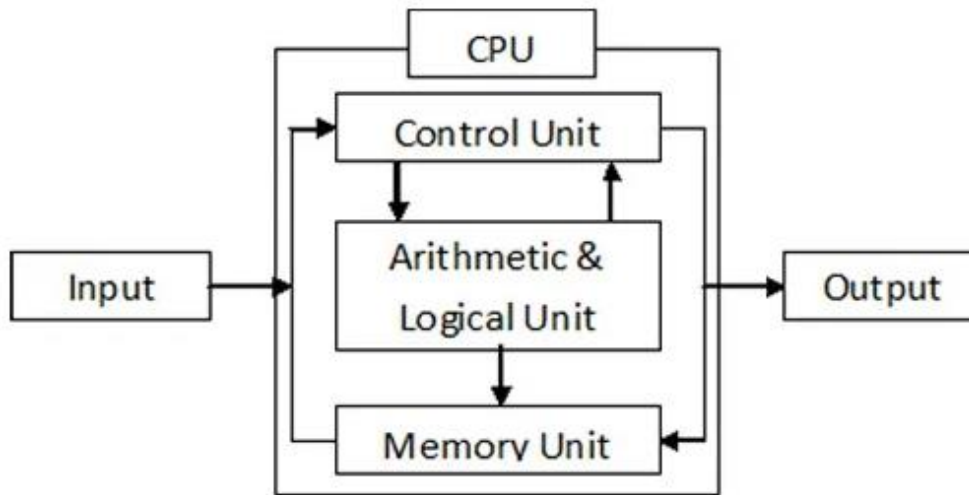


Fig: Block Diagram of Computer

Introduction to Computer Number System

Binary:

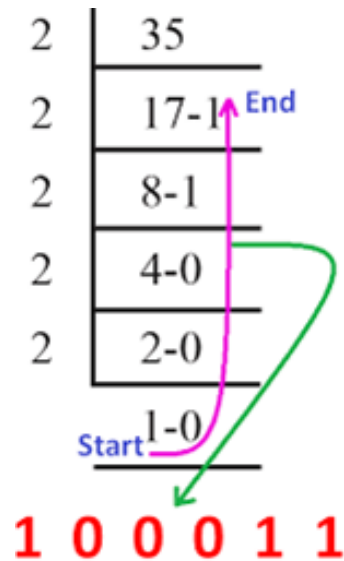


Fig: Decimal to binary

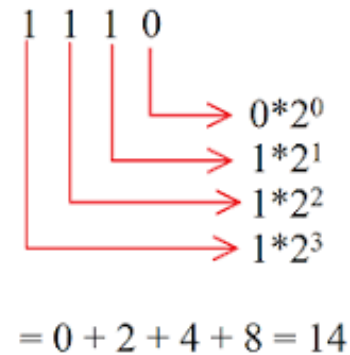


Fig: Binary to Decimal

Introduction to Computer Number System

Octal:

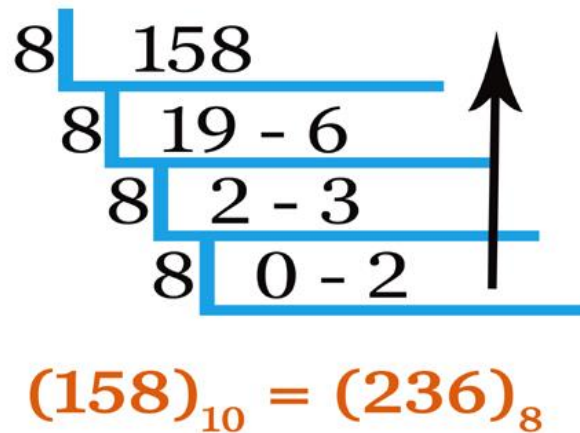


Fig: Decimal to Octal

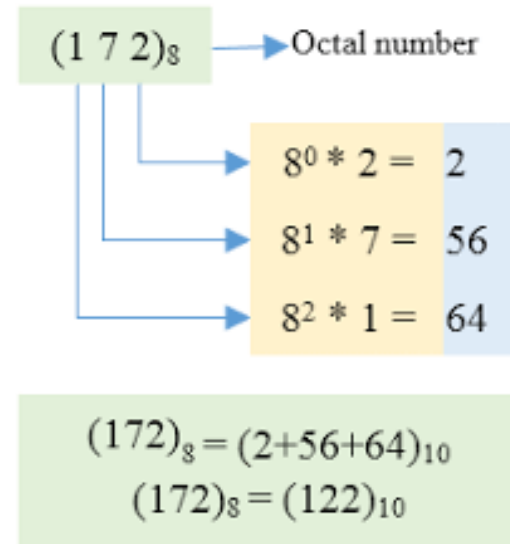


Fig: Octal to Decimal

Introduction to Computer Number System

Hexadecimal:

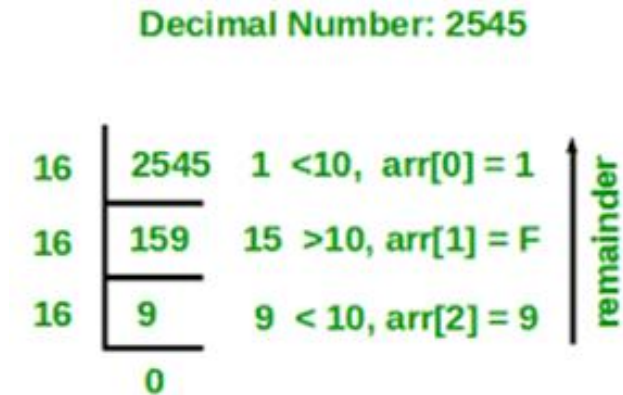


Fig: Decimal to Hexadecimal

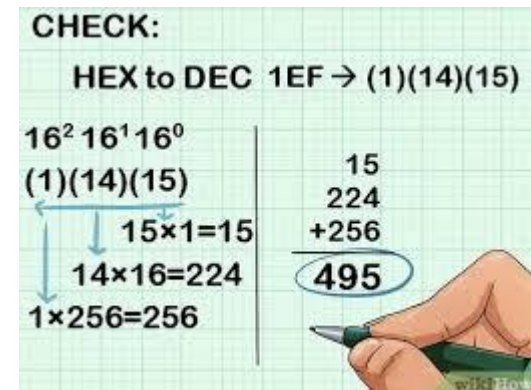


Fig: Hexadecimal to Decimal

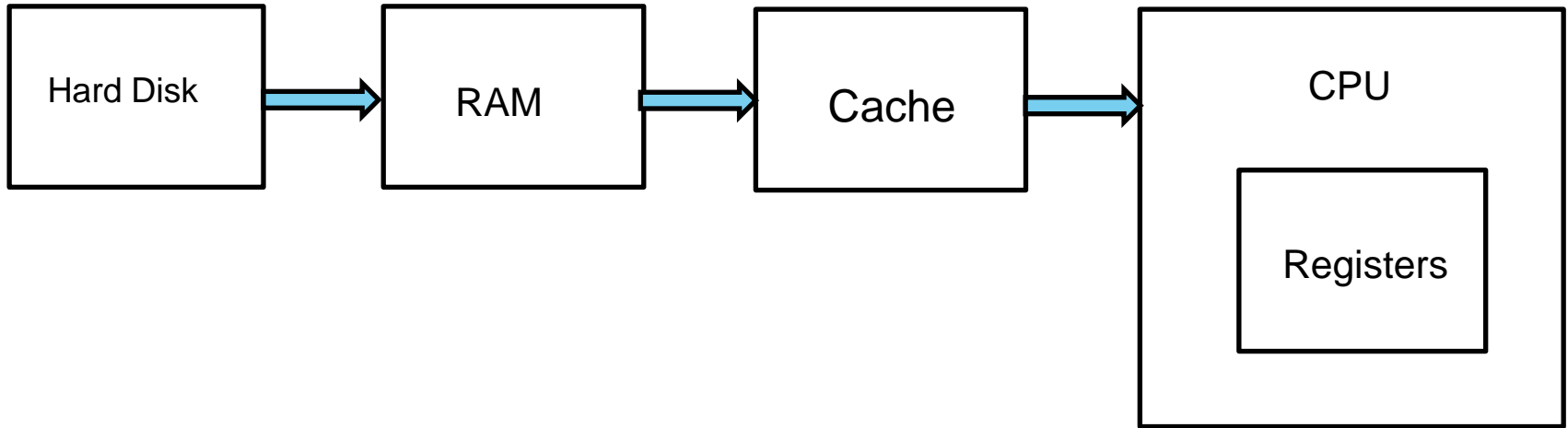
Introduction to Computer

Ascii Table

K2		✕ ✓ <i>fx</i>		=CHAR(J2)															
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1																			
2	1	␣	17	␣	33	!	49	1	65	A	81	Q	97	a	113	q			
3	2		18	␣	34	"	50	2	66	B	82	R	98	b	114	r			
4	3	␣	19	␣	35	#	51	3	67	C	83	S	99	c	115	s			
5	4	␣	20	␣	36	\$	52	4	68	D	84	T	100	d	116	t			
6	5	␣	21	␣	37	%	53	5	69	E	85	U	101	e	117	u			
7	6	␣	22	␣	38	&	54	6	70	F	86	V	102	f	118	v			
8	7	␣	23	␣	39	'	55	7	71	G	87	W	103	g	119	w			
9	8	␣	24	␣	40	(56	8	72	H	88	X	104	h	120	x			
10	9		25	␣	41)	57	9	73	I	89	Y	105	i	121	y			
11	10		26	␣	42	*	58	:	74	J	90	Z	106	j	122	z			
12	11	␣	27	␣	43	+	59	;	75	K	91	[107	k	123	{			
13	12	␣	28		44	,	60	<	76	L	92	\	108	l	124				
14	13		29		45	-	61	=	77	M	93]	109	m	125	}			
15	14	␣	30		46	.	62	>	78	N	94	^	110	n	126	~			
16	15	␣	31		47	/	63	?	79	O	95	_	111	o	127	␣			
17	16	␣	32		48	0	64	@	80	P	96	`	112	p	128	€			

Introduction to Computer

Data read and memory



Algorithms and FlowChart

Designing a Program

Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts to create models of programs.

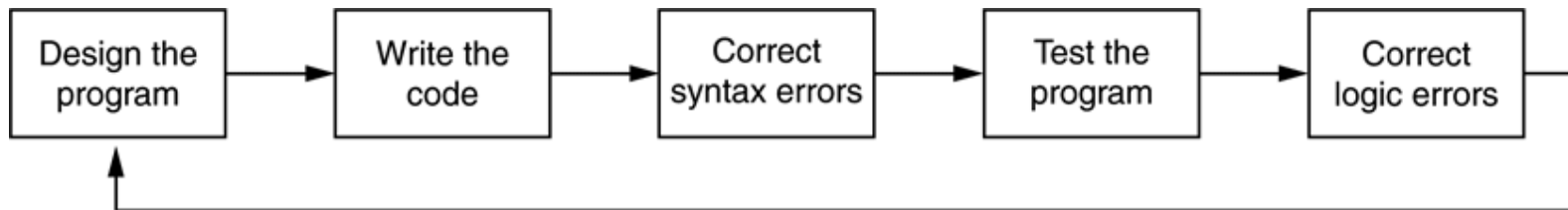


Fig: The Program Development Life Cycle

Algorithm and FlowChart

Pseudocode

Pseudocode is an informal language used to develop a program's design. It has no syntax rules and is not meant to be compiled or executed.

For Example:

Write a program to calculate and display the gross pay for an hourly paid employee.

Input the hours worked

Input the hourly pay rate

Calculate gross pay as hours worked multiplied by hourly pay rate

Display the gross pay

Algorithm and FlowChart

Flowchart

A **flowchart** is a tool that graphically depicts the steps that takes place in a program.

Types of Symbols:

Ovals—appear at the top or bottom of the flowchart, and are called terminal symbols

Parallelograms—used as input and output symbols

Rectangles—used as processing symbols

Arrows—represent the flow of the program

Algorithm and FlowChart

Flowchart

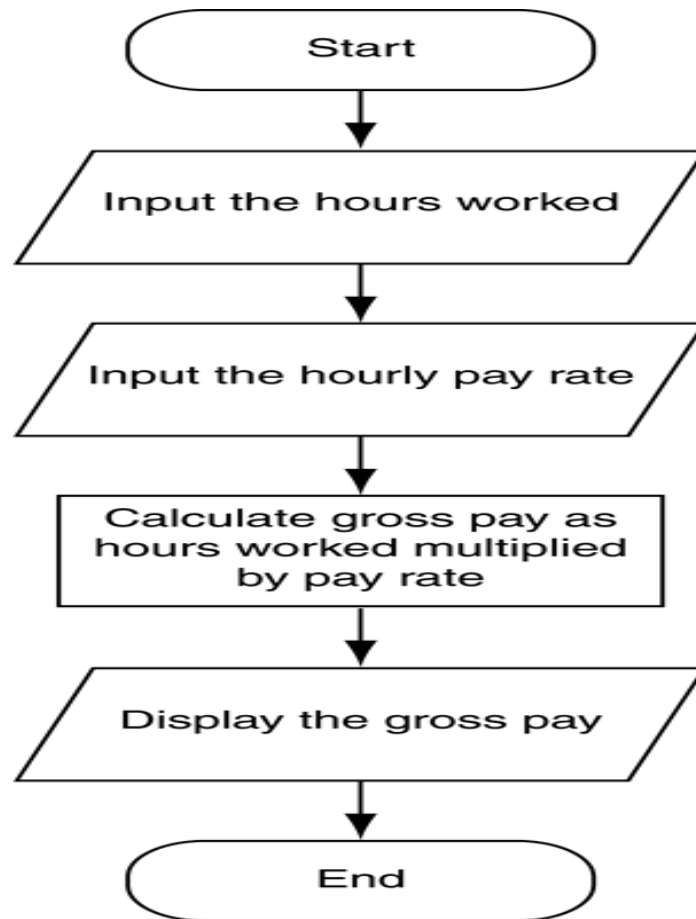
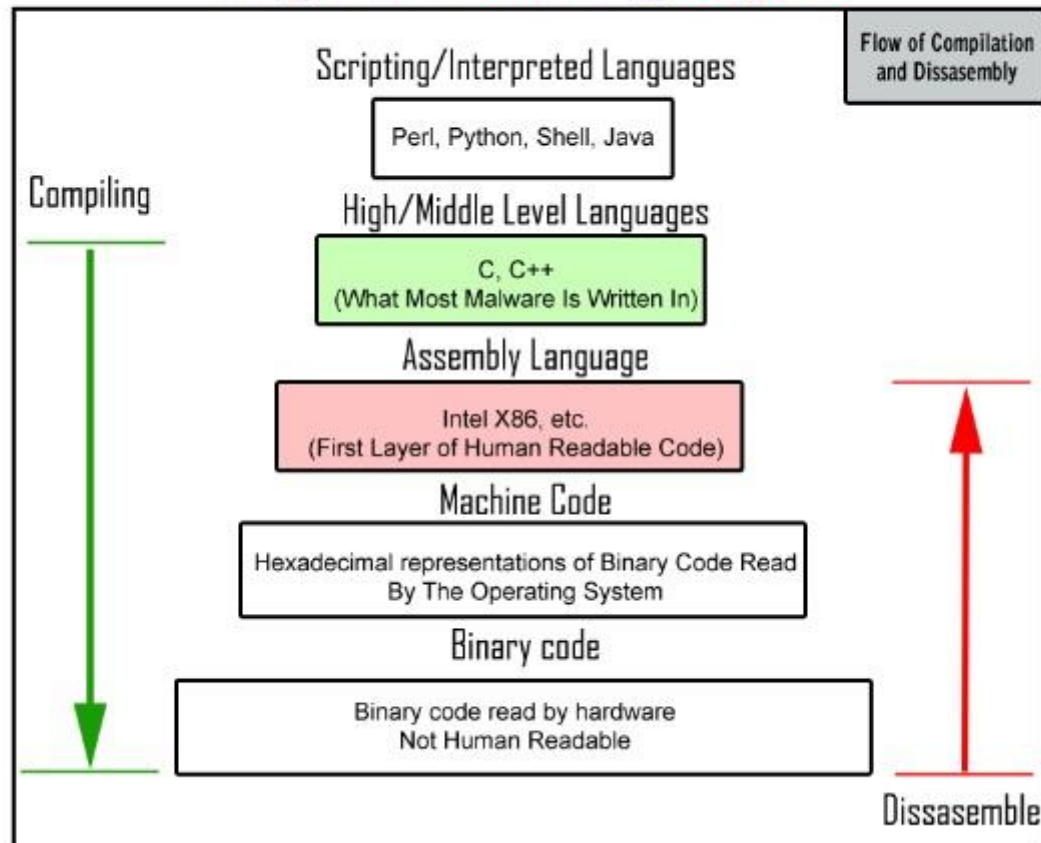


Fig: Flowchart for pay calculating program

Introduction to Programming

Language Representation

High Level Languages



Introduction to Programming

Choice of Programming Languages

- Web Applications : JavaScript, PHP, Ruby, HTML/CSS, Python
- Mobile Applications: Swift, Java, JavaScript, Objective-C
- Operating Systems: C, C++
- Distributed Systems: Go
- Enterprise Applications: Java, C#, C++, Erlang
- Analytics & Machine Learning: Python, R, Clojure, Julia
- Math & Scientific Computing: Matlab, Python, FORTRAN, ALGOL, APL, Julia, R, C++

Introduction to Programming

Compiler vs Interpreter

Compiler	Interpreter
Translates a High level program to machine language code - all at once.	Translates a High level program code to machine language code - one line of code at a time.
All errors found are displayed together after the compilation of the program is complete.	As translation and execution happens one line at a time so errors, if any are displayed for each line as it is translated to machine code.
Code execution and compilation of the code are two different processes.	Code Execution is a part of the code interpretation process.
Debugging a compiled program is relatively difficult as all the errors found are reported together.	Debugging an interpreted program is relatively easy, as one line at a time is executed and the error is reported.
Compiling a program is faster.	Interpreters are slower as compared to compilers.
C and C++ are examples of languages where the code written is compiled.	JavaScript, PHP, Ruby and Perl are examples of languages where the code written is interpreted.

Software Development Methodology

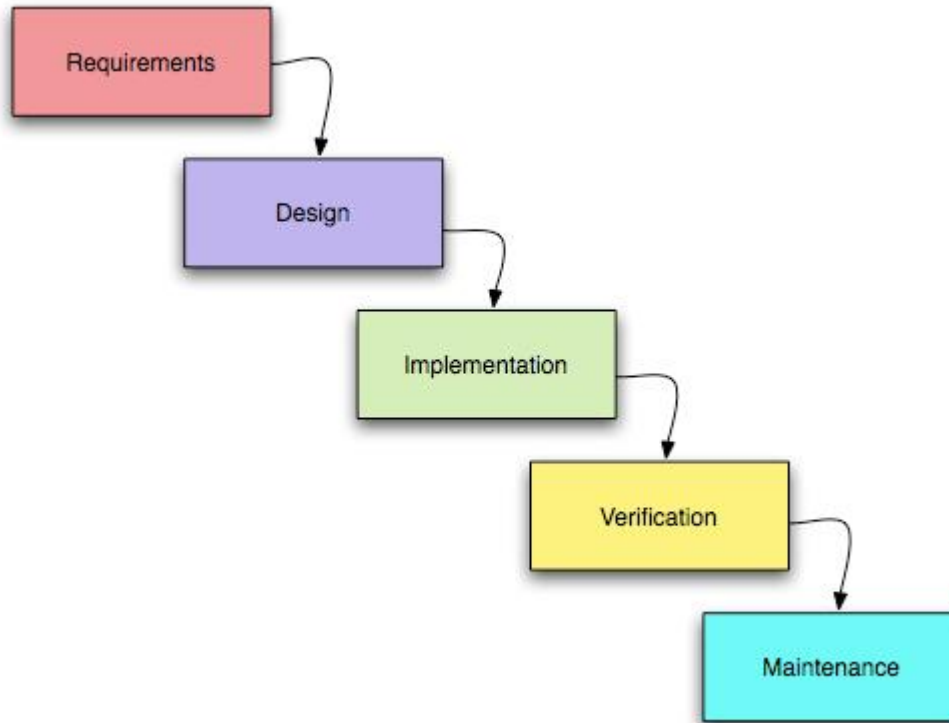


Fig: Waterfall Model

Software Development Methodology

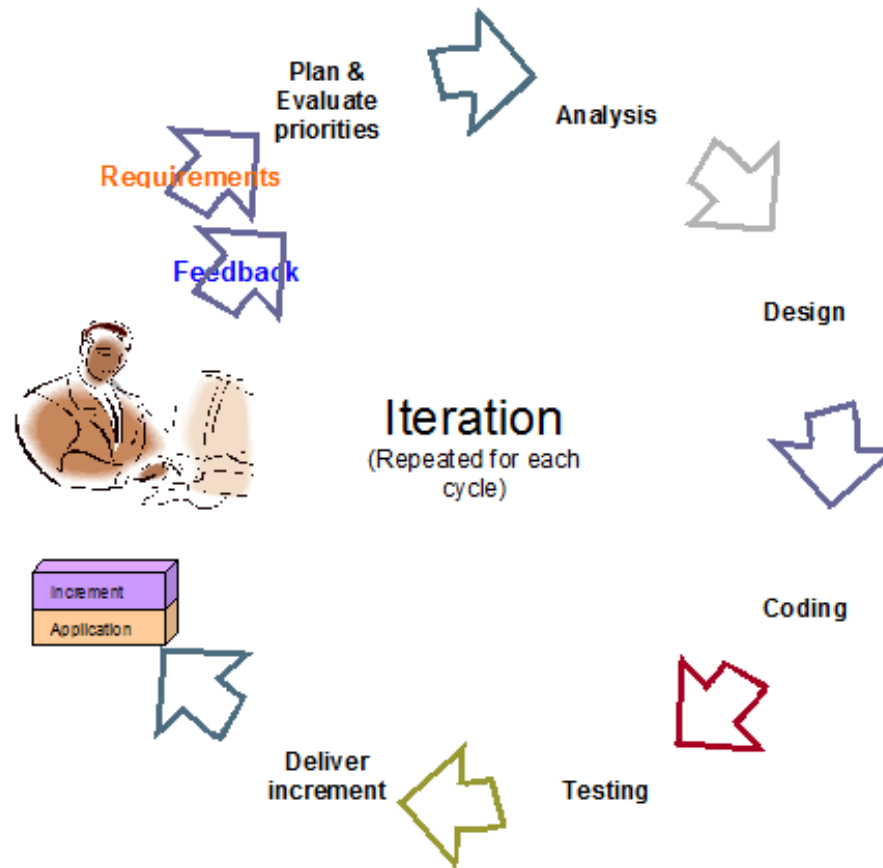


Fig: Agile Methodology

Operators in Programming Language

Operator:

A symbol, which indicates an operation to be performed. Operators are used to manipulate data in program.

Arithmetic Operators

Operator	Meaning
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Operators in Programming Language

Assignment Operators

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b

Operators in Programming Language

Relational Operators

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

Operators in Programming Language

Bitwise Operators

Operators	Meaning of operators
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Conditional and Loops

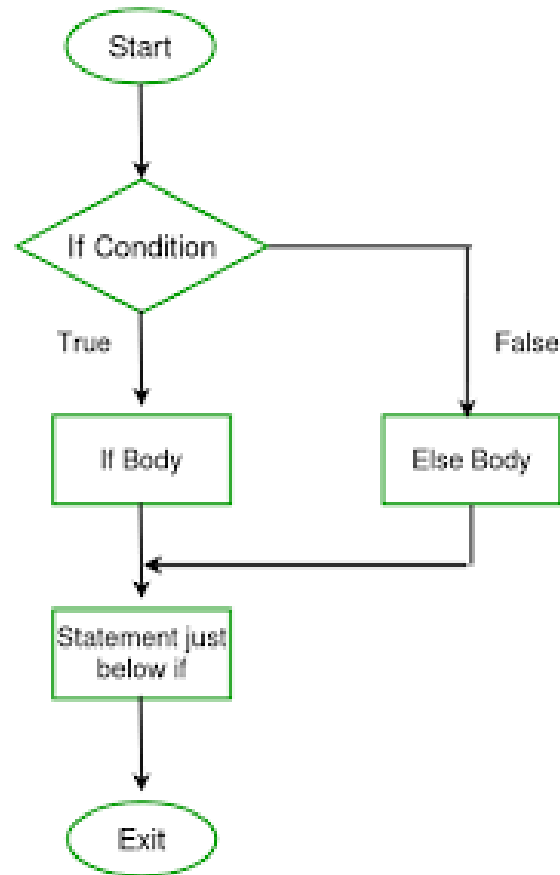
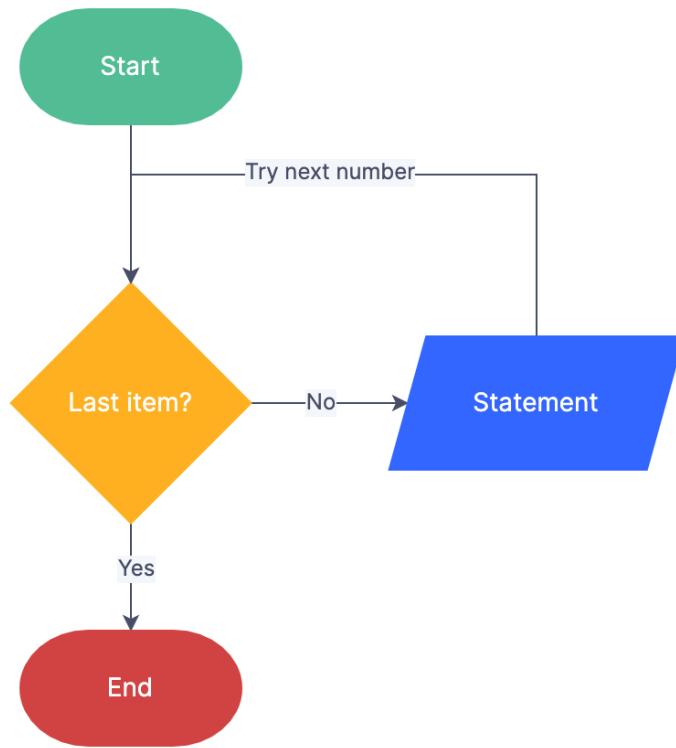


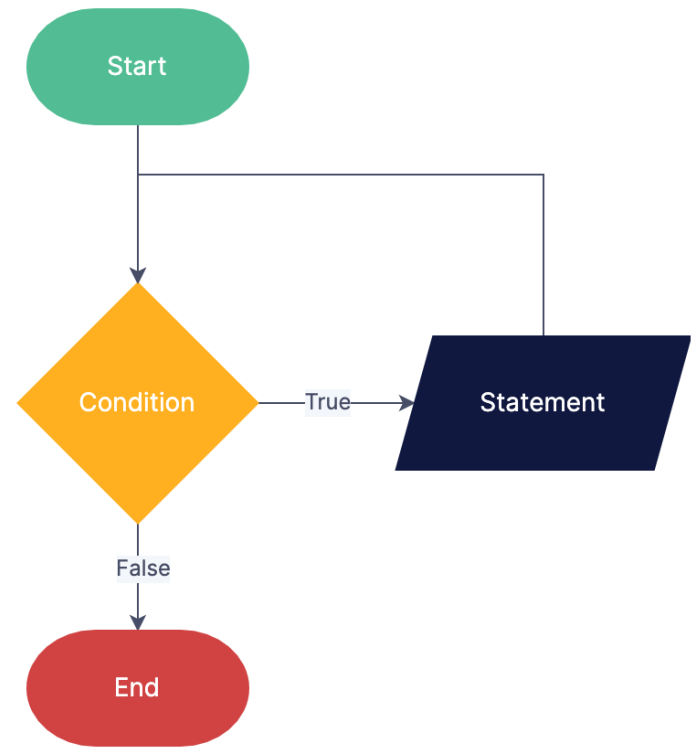
Fig: If/Else condition

Conditional and Loops

For Loop Flowchart



While Loop Flowchart



Python

Features

Created in 1991 by Guido Van Rossum

- Easy to learn and readable language
- Interpreted Language
- Dynamically typed language
- Open source and free
- Large standard library
- High-level language
- Object oriented programming language
- Large community support



Python

Applications

- Web development
- Game development
- Computer vision
- Machine learning
- GUI development
- Robotics
- Data analysis

Python

Organization using python

- Nasa
- IBM
- Google
- Amazon
- Facebook
- Instagram
- Spotify
- Quora
- Pinterest

Python

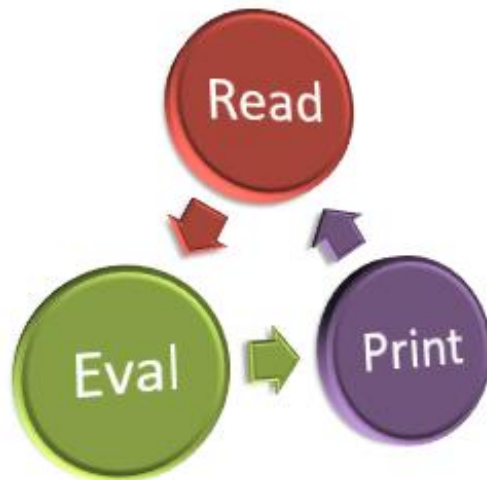
When you start Python, you will see something like:

Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37)

[MSC v.1934 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>



Python

The “>>>” is a Python *prompt* indicating that Python is ready for us to give it a command. These commands are called **statements**.

```
>>> print("Hello, world")  
Hello, world
```

```
>>> print(2+3)  
5
```

```
>>> print("2+3=", 2+3)  
2+3= 5
```

```
>>>
```

Python

Usually we want to execute several statements together that solve a common problem. One way to do this is to use a ***function***.

```
>>> def hello():  
    print("Hello")  
    print("Computers are Fun")
```

```
>>>
```

The first line tells Python we are *defining* a new function called hello.

The following lines are indented to show that they are part of the hello function.

Python

```
>>> def hello():  
    print("Hello")  
    print("Computers are Fun")
```

```
>>>
```

Notice that nothing has happened yet! We've defined the function, but we haven't told Python to perform the function!

A function is *invoked* by typing its name.

```
>>> hello()  
Hello  
Computers are Fun
```

```
>>>
```

Python

What's the deal with the ()'s?

Commands can have changeable parts called *parameters* that are placed between the ()'s.

```
>>> def greet(person):  
    print("Hello", person)  
    print("How are you?")
```

```
>>>
```

Python

```
>>> greet("Deepak")  
Hello Deepak  
How are you?
```

```
>>> greet("Seema")  
Hello Seema  
How are you?  
>>>
```

When we use parameters, we can customize the output of our function.

Python

When we exit the Python prompt, the functions we've defined cease to exist!

Programs are usually composed of functions, *modules*, or *scripts* that are saved on disk so that they can be used again and again.

A *module file* is a text file created in text editing software (saved as “plain text”) that contains function definitions.

A *programming environment* is designed to help programmers write programs and usually includes automatic indenting, highlighting, etc.

Python

```
# A simple program computing the sum of first n positive integers

def main():
    print("This program computes the sum of first n positive integers")
    n = eval(input('Enter an integer number greater than 0: '))
    total = 0
    for i in range(1, n+1):
        total = total + i
    print('Sum of first', n, 'positive integers is', total)

main()
```

We'll use *filename.py* when we save our work to indicate it's a Python program.

In this code we're defining a new function called **main**.

The `main()` at the end tells Python to run the code.

Inside a Python program

A simple program computing the sum of first n positive integers

Lines that start with *#* are called *comments*

Intended for human readers and ignored by Python

Python skips text from *#* to end of line

def main():

Beginning of the definition of a function called *main*

Since our program has only this one module, it could have been written without the *main* function.

The use of *main* is customary, however.

Inside a Python program

```
print("This program computes the sum of first n positive integers")
```

This line causes Python to print a message introducing the program.

```
n = eval(input('Enter an integer number greater than 0: '))
```

`n` is an example of a *variable*

A variable is used to assign a name to a value so that we can refer to it later.

The quoted information is displayed, and the number typed in response is stored in `n`.

Inside a Python Program

```
total = 0
```

total is another example of a *variable*

As the program will compute the sum of first n positive integers, we need an accumulator to accumulate the sum of the numbers.

The variable total is being initialised with value of 0.

Inside a Python Program

```
for i in range(1, n+1):
```

for is a *loop* construct

A loop tells Python to repeat the same thing over and over.

In this example, the following code will be repeated n times.

Inside a Python Program

```
total = total + i
```

This line is the *body* of the loop.

The body of the loop is what gets repeated each time through the loop.

The body of the loop is identified through **indentation**.

The effect of the loop is the same as repeating this line n times! (Depending on what is the integer value entered by user)

Inside a Python Program

Assuming that user enter 10 when prompted, i.e. $n = 10$

```
for i in range(1, 11):
```

```
    total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

```
total = total + i
```

These are equivalent!

Inside a Python Program

```
total = total + i
```

This is called an *assignment* statement

The part on the right-hand side (RHS) of the “=” is a mathematical expression.

+ is used to indicate addition

Once the value on the RHS is computed, it is stored back into (*assigned*) into total.

Inside a Python Program

```
print('Sum of first', n, 'positive integers is', total)
```

This line with the print function is outside the for loop. It is not indented, and thus is not inside the body of the loop.

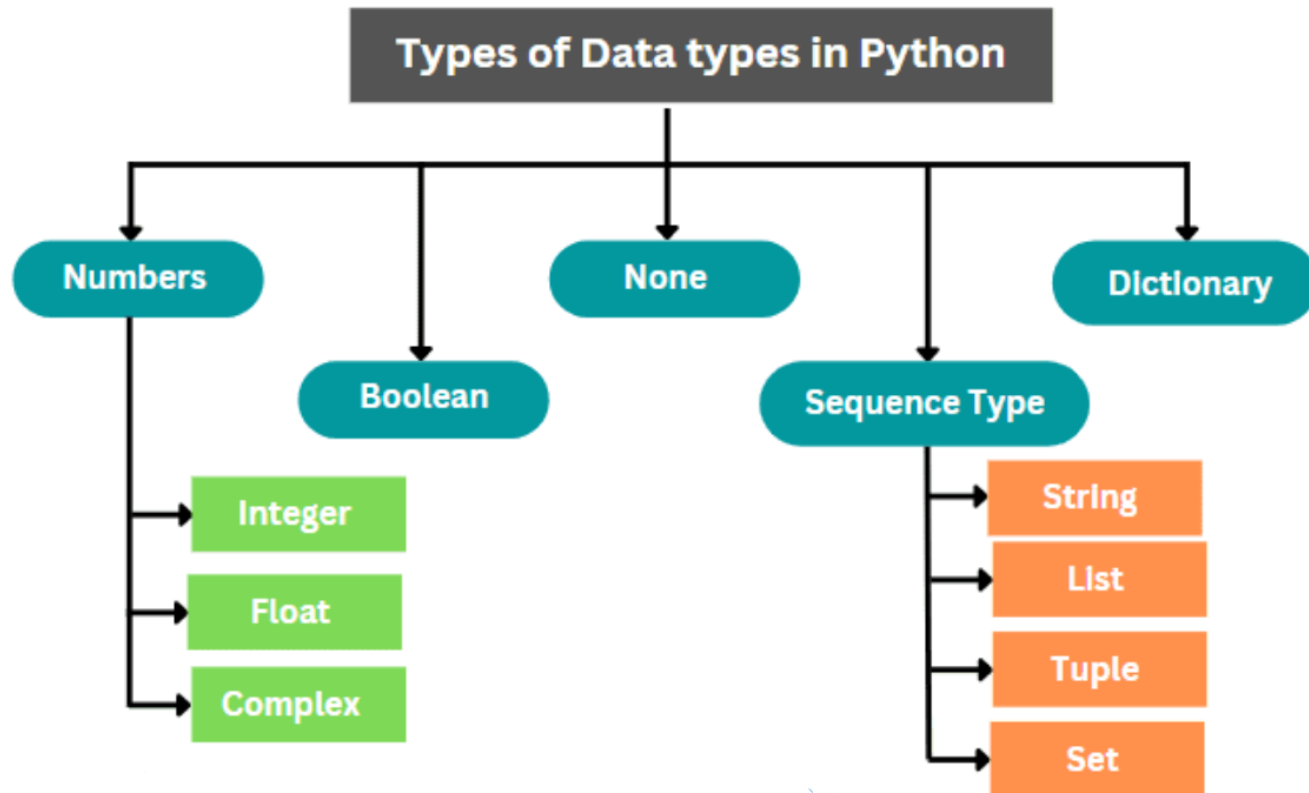
Once the loop is exited, this line will be executed – which display a message display the sum of the first n integers. (Note: The result depends on what is the value entered by user.)

```
main()
```

This last line tells Python to *execute* the code in the function *main*

Note: It is not part of the function *main*

Data Types



Data Types

```
iNum1 = 12
iNum2 = -100
fNum1 = 1.5
fNum2 = -1.5
bValue1 = True
bValue2 = False
str1 = 'James Bond is here'
str2 = '12345'
list1 = [1, 5, 7, 9]
list2 = [1.0, 5, 'String', True, 'Any Data']
tuple1 = (31, 1, 2020)
tuple2 = ('James', 123)
dict1 = {1:'One', 2:'Two', 3:'Three'}
dict2 = {'Jan':123.5, 'Feb': 25.6, 'Mar':95.0}
```

Data Types

```
print(iNum1, ': ', type(iNum1))
print(fNum1, ': ', type(fNum1))
print(bValue1, ': ', type(bValue1))
print(str1, ': ', type(str1))
print(list1, ': ', type(list1))
print(tuple1, ': ', type(tuple1))
print(dict1, ': ', type(dict1))
```

```
12 : <class 'int'>
1.5 : <class 'float'>
True : <class 'bool'>
James Bond is here : <class 'str'>
[1, 5, 7, 9] : <class 'list'>
(31, 1, 2020) : <class 'tuple'>
{1: 'One', 2: 'Two', 3: 'Three'} : <class 'dict'>
```

Data Types

Python has a special function to tell us the data type of any value.

```
>>> type(3)
<class 'int'>
```

```
>>> type(3.1)
<class 'float'>
```

```
>>> type(3.0)
<class 'float'>
```

```
>>> myInt = 32
>>> type(myInt)
<class 'int'>
>>>
```

Data Types

Operations on ints produce ints (except for /), operations on floats produce floats.

```
>>> 3.0+4.0
```

```
7.0
```

```
>>> 3+4
```

```
7
```

```
>>> 10.0/3.0
```

```
3.3333333333333335
```

```
>>> 10/3
```

```
3.3333333333333335
```

```
>>> 10 // 3
```

```
3
```

```
>>> 10.0 // 3.0
```

```
3.0
```

Data Types

Exercise

Try it out

$2 + 3 - 4$

$4 / 2 * 5$

$2 + 3 * 5$

$(2 + 3) * 5$

$2 + 3 * 5 ** 2$

$2 + 3 * 5 ** 2 - 1$

$-4 + 2$

Data Types

Example

```
>>> iNum = 3
>>> fNum = 3.5
>>> msg = 'This is a string'
>>> iNum * fNum
10.5
>>> iNum * msg # repeat string (iNum) times
'This is a stringThis is a stringThis is a string'
>>> msg + ' ' + msg # concatenation of string
'This is a string This is a string'
>>> fNum * msg # multiplication of string required the numeric to be integer
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    fNum * msg # multiplication of string required the numeric to be integer
TypeError: can't multiply sequence by non-int of type 'float'
>>> (msg + ' ') * iNum # integer could be in front or at the back of string
'This is a string This is a string This is a string '
>>>
```

Data Types

Type Conversions

In *mixed-typed expressions* Python will convert ints to floats.

Sometimes we want to control the type conversion. This is called *explicit typing*.

```
>>> float(15//4)
3.0
>>> int(3.9)
3
>>> int(3.1)
3
>>> round(3.5)
4
>>> round(1234.567, 2)
1234.57
>>> round(1234.567, -2)
1200.0
```

Math Library

Besides (+, -, *, /, //, **, %, abs), we have lots of other math functions available in a *math library*.

A *library* is a module with some useful definitions/functions.

Let's write a program to compute the roots of a quadratic equation!

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Math Library

To use a library, we need to make sure this line is in our program:

```
import math
```

Importing a library makes whatever functions are defined within it available to the program.

To access the sqrt library routine, we need to access it as *math.sqrt(x)*.

Using this dot notation tells Python to use the sqrt function found in the math library module.

To calculate the root, you can do

```
discRoot = math.sqrt(b*b - 4*a*c)
```

Math Library

```
# A program that computes the real roots of a quadratic equation.  
# Illustrates use of the math library.
```

```
import math # Makes the math library available.
```

```
def main():
```

```
    print("This program finds the real solutions to a quadratic")  
    print()
```

```
    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
```

```
    disc_root = math.sqrt(b * b - 4 * a * c)
```

```
    root1 = (-b + disc_root) / (2 * a)
```

```
    root2 = (-b - disc_root) / (2 * a)
```

```
    print()
```

```
    print("The solutions are:", root1, root2 )
```

```
main()
```

Factorial

Say you are waiting in a line with five other people. How many ways are there to arrange the six people?

720 is the factorial of 6 (abbreviated 6!)

Factorial is defined as:

$$n! = n(n-1)(n-2)\dots(1)$$

$$\text{So, } 6! = 6*5*4*3*2*1 = 720$$

Factorial

How did we calculate 6!?

$$6 * 5 = 30$$

Take that 30, and $30 * 4 = 120$

Take that 120, and $120 * 3 = 360$

Take that 360, and $360 * 2 = 720$

Take that 720, and $720 * 1 = 720$

Factorial

What's really going on?

We're doing repeated multiplications, and we're keeping track of the running product.

This algorithm is known as an *accumulator*, because we're building up or *accumulating* the answer in a variable, known as the *accumulator variable*.

The general form of an accumulator algorithm looks like this:

- Initialize the accumulator variable

- Loop until final result is reached

- Update the value of accumulator variable

Factorial

```
# factorial.py
# Program to compute the factorial of a number
# Illustrates for loop with an accumulator

def main():
    n = eval(input("Please enter a whole number: "))
    fact = 1

    for factor in range(n,1,-1):
        fact = fact * factor

    print("The factorial of", n, "is", fact)

main()
```

String

```
>>> str1="Hello"
```

```
>>> str2='spam'
```

```
>>> print(str1, str2)
```

```
Hello spam
```

```
>>> type(str1)
```

```
<class 'str'>
```

```
>>> type(str2)
```

```
<class 'str'>
```

String

Getting a string as input

```
>>> first_name = input("Please enter your name: ")
Please enter your name: Ram
>>> print("Hello", first_name)
Hello Ram
```

We can access the individual characters in a string through *indexing*.

The positions in a string are numbered from the left, starting with 0.

The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

String

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
```

```
>>> greet[0]
```

```
'H'
```

```
>>> print(greet[0], greet[2], greet[4])
```

```
H l o
```

```
>>> x = 8
```

```
>>> print(greet[x - 2])
```

```
B
```

String

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

In a string of n characters, the last character is at position $n-1$ since we start counting with 0.

We can index from the **right** side using **negative** indexes.

```
>>> greet[-1]
```

```
'b'
```

```
>>> greet[-3]
```

```
'B'
```

String

Indexing returns a string containing a **single** character from a larger string.

We can also access a **contiguous sequence** of characters, called a *substring*, through a process called *slicing*.

Slicing:

`<string>[<start>:<end>]`

start and end should both be ints.

The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.

String

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
```

```
'Hel'
```

```
>>> greet[5:9]
```

```
' Bob'
```

```
>>> greet[:5]
```

```
'Hello'
```

```
>>> greet[5:]
```

```
' Bob'
```

```
>>> greet[:]
```

```
'Hello Bob'
```

String

If either expression is missing, then the start or the end of the string are used.

Can we put two strings together into a longer string?

Concatenation “glues” two strings together (+)

Repetition builds up a string by multiple concatenations of a string with itself (*)

String

```
>>> "spam" + "eggs"
```

'spameggs'

```
>>> "Spam" + "And" + "Eggs"
```

'SpamAndEggs'

```
>>> 3 * "spam"
```

'spamspamspam'

```
>>> "spam" * 5
```

'spamspamspamspamspam'

```
>>> (3 * "spam") + ("eggs" * 5)
```

'spamspamspameggseggseggseggseggs'

String

The function *len* will return the length of a string.

```
>>> len("spam")
```

```
4
```

```
>>> for ch in "Spam!":  
    print (ch, end=" ")
```

```
S p a m !
```

String Formatting

String formatting is an easy way to get beautiful output!

Change Counter

Please enter the count of each coin type.

Quarters: 6

Dimes: 0

Nickels: 0

Pennies: 0

The total value of your change is 1.5

Shouldn't that be more like \$1.50??

String Formatting

We can format our output by modifying the print statement as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

Now we get something like:

The total value of your change is \$1.50

Key is the string format method.

```
<template-string>.format(<values>)
```

{ } within the template-string mark “slots” into which the values are inserted.

Each slot has description that includes *format specifier* telling Python how the value for the slot should appear.

String Formatting

If the width is wider than needed, numeric values are right-justified and strings are left-justified, by default.

You can also specify a justification before the width.

```
>>> "left justification: {0:<5}".format("Hi!")
```

```
'left justification: Hi! '
```

```
>>> "right justification: {0:>5}".format("Hi!")
```

```
'right justification:  Hi!'
```

```
>>> "centered: {0:^5}".format("Hi!")
```

```
'centered: Hi! '
```

String Formatting

A program to calculate the value of some change in dollars.

This version represents the total cash in cents.

```
def main():  
    print ("Change Counter\n")  
  
    print ("Please enter the count of each coin type.")  
    quarters = eval(input("Quarters: "))  
    dimes = eval(input("Dimes: "))  
    nickels = eval(input("Nickels: "))  
    pennies = eval(input("Pennies: "))  
    total = quarters * 25 + dimes * 10 + nickels * 5 + pennies  
  
    print ("The total value of your change is ${0}.{1:>2}"  
          .format(total//100, total%100))
```

String Formatting

```
>>> main()
```

Please enter the count of each coin type.

Quarters: 0

Dimes: 0

Nickels: 0

Pennies: 1

The total value of your change is \$0.01

```
>>> main()
```

Please enter the count of each coin type.

Quarters: 12

Dimes: 1

Nickels: 0

Pennies: 4

The total value of your change is \$3.14

Simple decisions

So far, we've viewed programs as sequences of instructions that are followed one after the other.

While this is a fundamental programming concept, it is not sufficient in itself to solve every problem. We need to be able to alter the sequential flow of a program to suit a particular situation.

Control structures allow us to alter this sequential program flow.

In this section, we'll learn about *decision structures*, which are statements that allow a program to execute different sequences of instructions for different cases, allowing the program to **choose** an appropriate course of action.

Simple decisions

Temperature Warning

A program to convert Celsius temps to Fahrenheit

```
def main():  
    celsius = eval(input("What is the Celsius temperature? "))  
    fahrenheit = 9/5 * celsius + 32  
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")  
  
main()
```

Let's say we want to modify that program to print a warning when the weather is extreme.

Any temperature over 90 degrees Fahrenheit and lower than 30 degrees Fahrenheit will cause a hot and cold weather warning, respectively.

Simple decisions

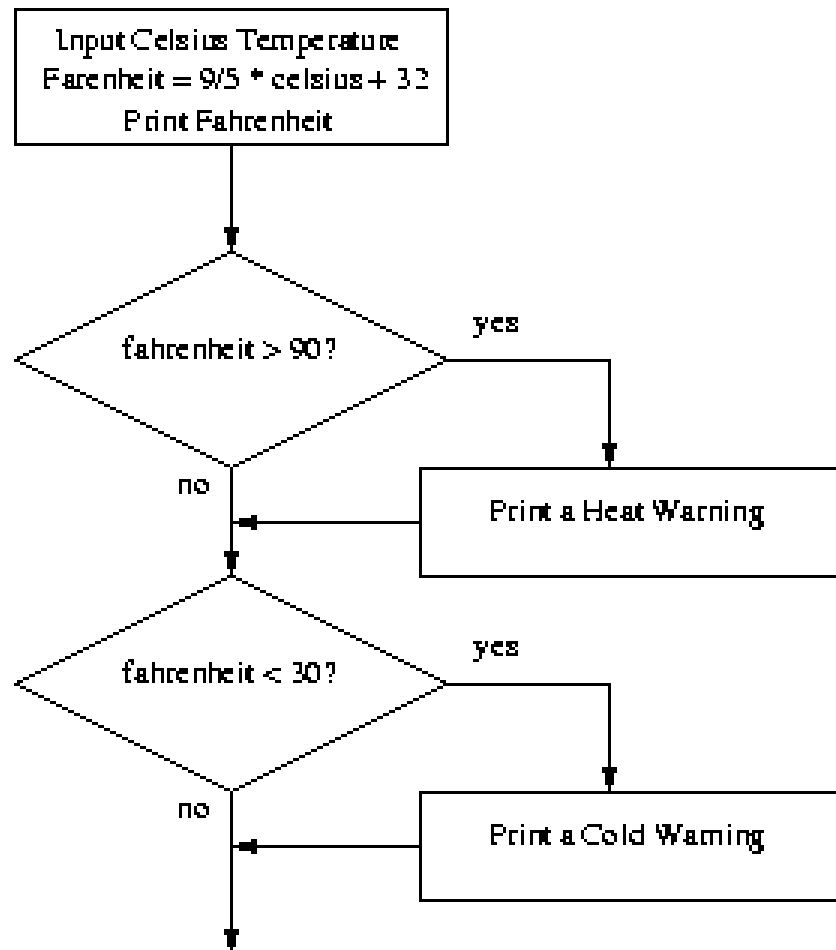
Temperature Warning

- Input the temperature in degrees Celsius (call it celsius)
- Calculate fahrenheit as $9/5 \text{ celsius} + 32$
- Output fahrenheit
- If fahrenheit > 90
 print a heat warning
- If fahrenheit < 30
 print a cold warning

This new algorithm has two *decisions* at the end. The indentation indicates that a step should be performed only if the condition listed in the previous line is true.

Decisions

Temperature Warning



Decisions

Temperature Warning

- # A program to convert Celsius temps to Fahrenheit.
- # This version issues heat and cold warnings.

```
def main():  
    celsius = eval(input("What is the Celsius temperature? "))  
    fahrenheit = 9 / 5 * celsius + 32  
    print("The temperature is", fahrenheit, "degrees fahrenheit.")  
  
    if fahrenheit > 90:  
        print("It's really hot out there, be careful!")  
    if fahrenheit < 30:  
        print("Brrrrr. Be sure to dress warmly")  
  
main()
```

Two-way decisions

In Python, a two-way decision can be implemented by attaching an `else` clause onto an `if` clause.

This is called an if-else statement:

```
if <condition>:  
    <statements>  
else:  
    <statements>
```

When Python first encounters this structure, it first evaluates the condition. If the condition is true, the statements under the `if` are executed.

If the condition is false, the statements under the `else` are executed.

In either case, the statements following the `if-else` are executed after either set of statements are executed.

Two-way decisions

A program checking whether a number is odd or even

```
def main():
```

```
    num = input("Enter a number: ")
```

```
    if num % 2 == 0:
```

```
        print("The entered number {} is even".format(num))
```

```
    else:
```

```
        print("The entered number {} is odd".format(num))
```

```
main()
```

Multi-way decisions

Imagine if we needed to make a five-way decision using nesting. The if-else statements would be nested four levels deep!

There is a construct in Python that achieves this, combining an else followed immediately by an if into a single elif.

```
if <condition1>:  
    <case1 statements>  
elif <condition2>:  
    <case2 statements>  
elif <condition3>:  
    <case3 statements>  
...  
else:  
    <default statements>
```

Multi-way decisions

This form sets of any number of mutually exclusive code blocks.

Python evaluates each condition in turn looking for the first one that is true. If a true condition is found, the statements indented under that condition are executed, and control passes to the next statement after the entire if-elif-else.

If none are true, the statements under else are performed.

The else is optional. If there is no else, it's possible no indented block would be executed.

Multi-way design

Type of character

A program demonstrating use of nested if else

Check the type of a character

```
_char = input("Enter any character: ")  
print("You have entered", _char)
```

Check for alphabet, digit or special char

```
if _char[0].isalpha():
```

```
    print(_char[0] + " is an alphabet")
```

```
elif _char[0].isdigit() :
```

```
    print(_char[0]+ " is a digit")
```

```
else :
```

```
    print(_char[0]+ " is a special character.")
```

Exception Handling

```
# quadratic1.py  
# A program that computes the real roots of a quadratic equation.  
# Note: This program crashes if the equation has no real roots.
```

```
import math
```

```
def main():
```

```
    print("This program finds the real solutions to a quadratic")
```

```
    a, b, c = eval(input("\nPlease enter the coefficients (a, b, c): "))
```

```
    disc_root = math.sqrt(b * b - 4 * a * c)
```

```
    root1 = (-b + disc_root) / (2 * a)
```

```
    root2 = (-b - disc_root) / (2 * a)
```

```
    print("\nThe solutions are:", root1, root2)
```

Exception Handling

As per the comment, when $b^2 - 4ac < 0$, the program crashes.
This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 1, 1, 2

Traceback (most recent call last):

File "C:\Users\pradeep\Desktop\bla.py", line 15, in <module>
 main()

File "C:\Users\pradeep\Desktop\bla.py", line 9, in main
 disc_root = math.sqrt(b * b - 4 * a * c)

ValueError: math domain error

Exception Handling

```
# quadratic2.py
# A program that computes the real roots of a quadratic equation.
# Bad version using a simple if to avoid program crash

import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discrim = b * b - 4 * a * c

    if discrim >= 0:
        disc_root = math.sqrt(discrim)
        root1 = (-b + disc_root) / (2 * a)
        root2 = (-b - disc_root) / (2 * a)
        print("\nThe solutions are:", root1, root2)
```

Exception Handling

We first calculate the discriminant (b^2-4ac) and then check to make sure it's nonnegative. If it is, the program proceeds and we calculate the roots.

Look carefully at the program. What's wrong with it? Hint: What happens when there are no real roots?

This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 1,1,1

>>>

This is almost worse than the version that crashes, because we don't know what went wrong!

Exception Handling

```
# quadratic3.py
# A program that computes the real roots of a quadratic equation.
# Illustrates use of a two-way decision
```

```
import math
```

```
def main():
```

```
    print("This program finds the real solutions to a quadratic\n")
```

```
    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
```

```
    discrim = b * b - 4 * a * c
```

```
    if discrim < 0:
```

```
        print("\nThe equation has no real roots!")
```

```
    else:
```

```
        disc_root = math.sqrt(b * b - 4 * a * c)
```

```
        root1 = (-b + disc_root) / (2 * a)
```

```
        root2 = (-b - disc_root) / (2 * a)
```

```
        print ("\nThe solutions are:", root1, root2 )
```

Exception Handling

>>>

This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 1,1,2

The equation has no real roots!

>>>

This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 2, 5, 2

The solutions are: -0.5 -2.0

Exception Handling

The newest program is great, but it still has some quirks!

This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 1,2,1

The solutions are: -1.0 -1.0

While correct, this method might be confusing for some people. It looks like it has mistakenly printed the same number twice!

Double roots occur when the discriminant is exactly 0, and then the roots are $-b/2a$.

It looks like we need a three-way decision!

Exception Handling

```
# quadratic4.py
# Illustrates use of a multi-way decision
```

```
import math
```

```
def main():
```

```
    print("This program finds the real solutions to a quadratic\n")
    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
```

```
    discrim = b * b - 4 * a * c
```

```
    if discrim < 0:
        print("\nThe equation has no real roots!")
```

```
    elif discrim == 0:
        root = -b / (2 * a)
        print("\nThere is a double root at", root)
```

```
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
```

Exception Handling

In the quadratic program we used decision structures to avoid taking the square root of a negative number, thus avoiding a run-time error.

This is true for many programs: decision structures are used to protect against rare but possible errors.

Sometimes programs get so many checks for special cases that the algorithm becomes hard to follow.

Programming language designers have come up with a mechanism to handle *exception handling* to solve this design problem.

The programmer can write code that catches and deals with errors that arise while the program is running, i.e., “Do these steps, and if any problem crops up, handle it this way.”

This approach obviates the need to do explicit checking at each step in the algorithm.

Exception Handling

```
# quadratic5.py
# A program that computes the real roots of a quadratic equation.
# Illustrates exception handling to avoid crash on bad inputs
```

```
import math
```

```
def main():
```

```
    print("This program finds the real solutions to a quadratic\n")
```

```
    try:
```

```
        a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
```

```
        disc_root = math.sqrt(b * b - 4 * a * c)
```

```
        root1 = (-b + disc_root) / (2 * a)
```

```
        root2 = (-b - disc_root) / (2 * a)
```

```
        print("\nThe solutions are:", root1, root2)
```

```
    except ValueError:
```

```
        print("\nNo real roots")
```

Exception Handling

The try statement has the following form:

```
try:  
    <body>  
except <ErrorType>:  
    <handler>
```

When Python encounters a try statement, it attempts to execute the statements inside the body.

If there is no error, control passes to the next statement after the try...except.

If an error occurs while executing the body, Python looks for an except clause with a matching error type. If one is found, the handler code is executed.

Exception Handling

The original program generated this error with a negative discriminant:

Traceback (most recent call last):

```
File "C:\Users\pradeep\Desktop\bla.py", line 15, in <module>  
    main()
```

```
File "C:\Users\pradeep\Desktop\bla.py", line 9, in main  
    disc_root = math.sqrt(b * b - 4 * a * c)
```

```
ValueError: math domain error
```

The last line, “ValueError: math domain error”, indicates the specific type of error.

Exception Handling

The `try...except` can be used to catch *any* kind of error and provide for a graceful exit.

In the case of the quadratic program, other possible errors include not entering the right number of parameters (“unpack tuple of wrong size”), entering an identifier instead of a number (`NameError`), entering an invalid Python expression (`TypeError`).

A single `try` statement can have multiple `except` clauses.

Exception Handling

```
# quadratic6.py
```

```
import math
```

```
print("This program finds the real solutions to a quadratic\n")
```

```
try:
```

```
    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))
```

```
    disc_root = math.sqrt(b * b - 4 * a * c)
```

```
    root1 = (-b + disc_root) / (2 * a)
```

```
    root2 = (-b - disc_root) / (2 * a)
```

```
    print("\nThe solutions are:", root1, root2 )
```

Exception Handling

quadratic6.py

```
except ValueError as excObj:
    if str(excObj) == "math domain error":
        print("No Real Roots")
    else:
        print("You didn't give me the right number of coefficients.")
except NameError:
    print("\nYou didn't enter three numbers.")
```


Exception Handling

quadratic6.py

except TypeError:

print("\nYour inputs were not all numbers.")

except SyntaxError:

print("\nYour input was not in the correct form. Missing comma?")

except:

print("\nSomething went wrong, sorry!")

Exception Handling

The multiple `excepts` act like `elifs`. If an error occurs, Python will try each `except` looking for one that matches the type of error.

The bare `except` at the bottom acts like an `else` and catches any errors without a specific match.

If there was no bare `except` at the end and none of the `except` clauses match, the program would still crash and report an error.

Exception Handling

Division by zero

Program demonstrating exception handling for zero division error

```
num1 = 5
```

```
num2 = 0
```

```
result = num1 / num2
```

```
print(result)
```

Exception handled for zero division

try:

```
    result = num1/num2
```

```
    print(result)
```

```
except FileNotFoundError as e:
```

```
    print(e)
```

```
except ZeroDivisionError as e:
```

```
    print(e)
```

```
except Exception:
```

```
    print("Trying to divide by zero.")
```

Loops


The for statement allows us to iterate through a sequence of values.


```
for <var> in <sequence>:  
    <body>
```


The loop index variable `var` takes on each successive value in the sequence, and the statements in the body of the loop are executed once for each value.


Loops


For loop

1st iteration: 
`for num in [1, 2, 3, 4, 5]:`
`print(num)`

2nd iteration: 
`for num in [1, 2, 3, 4, 5]:`
`print(num)`

3rd iteration: 
`for num in [1, 2, 3, 4, 5]:`
`print(num)`

4th iteration: 
`for num in [1, 2, 3, 4, 5]:`
`print(num)`

5th iteration: 
`for num in [1, 2, 3, 4, 5]:`
`print(num)`

Loops

Suppose we want to write a program that can compute the average of a series of numbers entered by the user.

To make the program general, it should work with any size set of numbers.

We don't need to keep track of each number entered, we only need know the running sum and how many numbers have been added.

We've run into some of these things before!

A series of numbers could be handled by some sort of loop. If there are n numbers, the loop should execute n times.

We need a running sum. This will use an ***accumulator***.

Loops

- Input the count of the numbers, n
- Initialize sum to 0
- Loop n times
 - Input a number, x
 - Add x to sum
- Output average as sum/n

Loops

Example

```
# average1.py
# A program to average a set of numbers
# Illustrates counted loop with accumulator
```

```
def main():
    n = eval(input("How many numbers do you have? "))
    sum = 0.0
    for i in range(n):
        x = eval(input("Enter a number >> "))
        sum = sum + x
    print("\nThe average of the numbers is", sum / n)
```

Note: that sum is initialized to 0.0 so that sum/n returns a float!

Loops

Example

How many numbers do you have? 5

Enter a number >> 32

Enter a number >> 45

Enter a number >> 34

Enter a number >> 76

Enter a number >> 45

The average of the numbers is 46.4

Loops

Example

Analysis

Money deposited in a bank account earns interest.

How much will the account be worth 10 years from now?

Inputs: principal, interest rate

Output: value of the investment in 10 years

Loops

Example

```
# A program to compute the value of an investment  
# carried 10 years into the future
```

```
def main():  
    print('This program calculates the future value of a ' + \  
          '10-year investment.')  
  
    principal = eval(input('Enter the initial principal: '))  
    i = eval(input('Enter the annual interest rate: '))  
  
    for _ in range(10):  
        principal = principal * (1 + i)  
  
    print('The value in 10 years is:', principal)
```

Loops

Example

```
>>> main()
```

This program calculates the future value of a 10-year investment.

Enter the initial principal: 100

Enter the annual interest rate: .10

The value in 10 years is: 259.37424601000026

```
>>> main()
```

This program calculates the future value of a 10-year investment.

Enter the initial principal: 100

Enter the annual interest rate: .03

The value in 10 years is: 134.39163793441222

```
>>>
```

Indefinite Loop

That last program got the job done, but you need to know ahead of time how many numbers you'll be dealing with.

What we need is a way for the computer to take care of counting how many numbers there are.

The for loop is a *definite* loop, meaning that the number of iterations is determined when the loop starts.

We can't use a definite loop unless we know the number of iterations ahead of time. We can't know how many iterations we need until all the numbers have been entered.

We need another tool!

The *indefinite* or *conditional* loop keeps iterating until certain conditions are met.

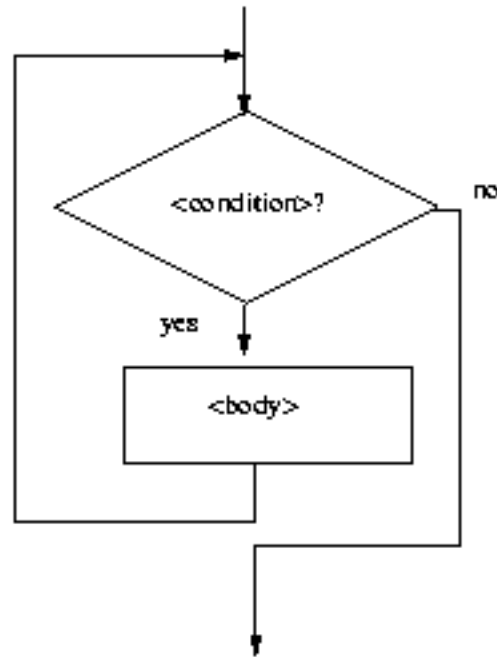
Indefinite Loop

```
while <condition>:  
    <body>
```

condition is a Boolean expression, just like in if statements. The body is a sequence of one or more statements.

Semantically, the body of the loop executes repeatedly as long as the condition remains true. When the condition is false, the loop terminates.

Indefinite Loop



The condition is tested at the top of the loop. This is known as a *pre-test* loop. If the condition is initially false, the loop body will not execute at all.

Indefinite Loop

Here's an example of a while loop that counts from 0 to 10:

```
i = 0
while i <= 10:
    print(i)
    i = i + 1
```

The code has the same output as this for loop:

```
for i in range(11):
    print(i)
```


Indefinite Loop

The while statement is simple, but yet powerful and dangerous – they are a common source of program errors.

```
i = 0  
while i <= 10:  
    print(i)
```

What happens with this code?

Indefinite Loop

Example

```
# average1.py
# A program to average a set of numbers
# Illustrates interactive loop with two accumulators
```

```
def main():
    more_data = "yes"
    sum = 0.0
    count = 0
    while more_data[0] == 'y':
        x = eval(input("Enter a number >> "))
        sum = sum + x
        count = count + 1
        more_data = input("Do you have more numbers (yes or no)? ")
    print("\nThe average of the numbers is", sum / count)
```

Indefinite Loop

Example

Enter a number >> 32
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes or no)? yes
Enter a number >> 34
Do you have more numbers (yes or no)? yup
Enter a number >> 76
Do you have more numbers (yes or no)? y
Enter a number >> 45
Do you have more numbers (yes or no)? nah

The average of the numbers is 46.4

Indefinite Loop

Exercise

Write a while loop that asks the user to enter two numbers. The numbers should be added and the sum displayed. The loop should ask the user if he or she wishes to perform the operation again. If so, the loop should repeat, otherwise it should terminate.

Indefinite Loop

```
# average2.py
# A program to average a set of numbers
# Illustrates sentinel loop using negative input as sentinel
```

```
def main():
    sum = 0.0
    count = 0
    x = eval(input("Enter a number (negative to quit) >> "))
    while x >= 0:
        sum = sum + x
        count = count + 1
        x = eval(input("Enter a number (negative to quit) >> "))
    print("\nThe average of the numbers is", sum / count)
```

Indefinite Loop

Enter a number (negative to quit) >> 32

Enter a number (negative to quit) >> 45

Enter a number (negative to quit) >> 34

Enter a number (negative to quit) >> 76

Enter a number (negative to quit) >> 45

Enter a number (negative to quit) >> -1

The average of the numbers is 46.4

Indefinite Loop

This version provides the ease of use of the interactive loop without the hassle of typing ‘y’ all the time.

There’s still a shortcoming – using this method we can’t average a set of positive *and negative* numbers.

If we do this, our sentinel can no longer be a number.

We could input all the information as strings.

Valid input would be converted into numeric form. Use a character-based sentinel.

We could use the *empty string* (“”)!

Indefinite Loop

```
# average4.py
# A program to average a set of numbers
# Illustrates sentinel loop using empty string as sentinel
```

```
def main():
    sum = 0.0
    count = 0
    xStr = input("Enter a number (<Enter> to quit) >> ")
    while xStr != "":
        x = eval(xStr)
        sum = sum + x
        count = count + 1
        xStr = input("Enter a number (<Enter> to quit) >> ")
    print("\nThe average of the numbers is", sum / count)
```


Indefinite Loop

```
Enter a number (<Enter> to quit) >> 34
Enter a number (<Enter> to quit) >> 23
Enter a number (<Enter> to quit) >> 0
Enter a number (<Enter> to quit) >> -25
Enter a number (<Enter> to quit) >> -34.4
Enter a number (<Enter> to quit) >> 22.7
Enter a number (<Enter> to quit) >>
```

The average of the numbers is 3.38333333333

Functions

So far, we've seen three different types of functions:
Our programs comprise a single function called `main()`.

Built-in Python functions (`input`, `print`).

Functions from the standard libraries (`math.sqrt`).

Having similar or identical code in more than one place has some drawbacks.

- Issue one: writing the same code twice or more.
- Issue two: This same code must be maintained in two separate places.

Functions

Functions can be used to reduce code duplication and make programs more easily understood and maintained.

A function is like a *subprogram*, a small program inside of a program.

The basic idea – we write a sequence of statements and then give that sequence a name. We can then execute this sequence at any time by referring to the name.

The part of the program that creates a function is called a *function definition*.

When the function is used in a program, we say the definition is *called* or *invoked*.

Functions

Benefits

- Simpler code
- Code reuse
- Better testing
- Faster development
- Easier facilitation of teamwork

Functions

Happy Birthday lyrics...

```
def main():  
    print("Happy birthday to you!")  
    print("Happy birthday to you!")  
    print("Happy birthday, dear Fred...")  
    print("Happy birthday to you!")
```

Gives us this...

```
>>> main()  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred...  
Happy birthday to you!
```

Functions

There's some duplicated code in the program! (`print("Happy birthday to you!")`)

We can define a function to print out this line:

```
def happy():  
    print("Happy birthday to you!")
```

With this function, we can rewrite our program.

Functions

The new program –

```
def sing_fred():  
    happy()  
    happy()  
    print("Happy birthday, dear Fred...")  
    happy()
```

Gives us this output –

```
>>> sing_fred()  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred...  
Happy birthday to you!
```

Functions

Creating this function saved us a lot of typing!

What if it's Lucy's birthday? We could write a new `sing_lucy` function!

```
def sing_lucy():  
    happy()  
    happy()  
    print("Happy birthday, dear Lucy...")  
    happy()
```


Functions

We could write a main program to sing to both Lucy and Fred

```
def main():  
    sing_fred()  
    print()  
    sing_lucy()
```

This gives us this new output

```
>>> main()
```

```
Happy birthday to you!
```

```
Happy birthday to you!
```

```
Happy birthday, dear Fred..
```

```
Happy birthday to you!
```

```
Happy birthday to you!
```

```
Happy birthday to you!
```

```
Happy birthday, dear Lucy...
```

```
Happy birthday to you!
```

Functions

This is working great! But... there's still a lot of code duplication.

The only difference between `sing_fred` and `sing_lucy` is the name in the third print statement.

These two routines could be collapsed together by using a *parameter*.

Functions

The generic function sing

```
def sing(person):  
    happy()  
    happy()  
    print("Happy birthday, dear", person + ".")  
    happy()
```

This function uses a parameter named *person*. A *parameter* is a variable that is initialized when the function is called.

Functions

Our new output –

```
>>> sing("Fred")
```

```
Happy birthday to you!
```

```
Happy birthday to you!
```

```
Happy birthday, dear Fred.
```

```
Happy birthday to you!
```

We can put together a new main program!

Functions

Our new main program:

```
def main():  
    sing("Fred")  
    print()  
    sing("Lucy")
```

Gives us this output:

```
>>> main()  
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Fred.  
Happy birthday to you!
```

```
Happy birthday to you!  
Happy birthday to you!  
Happy birthday, dear Lucy.  
Happy birthday to you!
```

Functions

A function definition looks like this:

```
def <name>(<formal-parameters>):  
    <body>
```

The name of the function must be an identifier

Formal-parameters is a *possibly empty list* of variable names.

Formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body.

Functions

A function is called by using its name followed by a list of *actual parameters* or *arguments*.

<name>(<actual-parameters>)

When Python comes to a function call, it initiates a four-step process.

- The calling program suspends execution at the point of the call.
- The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
- The body of the function is executed.
- Control returns to the point just after where the function was called.

Functions

A function is called by using its name followed by a list of *actual parameters* or *arguments*.

<name>(<actual-parameters>)

When Python comes to a function call, it initiates a four-step process.

- The calling program suspends execution at the point of the call.
- The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
- The body of the function is executed.
- Control returns to the point just after where the function was called.

Functions

Let's trace through the following code:

```
sing("Fred")  
print()  
sing("Lucy")
```


When Python gets to `sing("Fred")`, execution of main is temporarily suspended.

Python looks up the definition of `sing` and sees that it has one formal parameter, `person`.

Functions

The formal parameter is assigned the value of the actual parameter. It's as if the following statement had been executed:

person = "Fred"



The diagram illustrates the process of argument passing. A blue arrow originates from the `sing("Fred")` call within the `main` function and points to the `def sing(person):` definition, indicating that the string "Fred" is being passed as the argument for the `person` parameter.

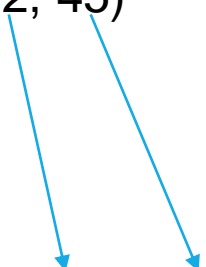
```
def main():  
    sing("Fred")  
    print()  
    sing("Lucy")  
  
def sing(person):  
    happy()  
    happy()  
    print("Happy Birthday, dear", person + ".")  
    happy()  
  
person: "Fred"
```

Functions

Passing Multiple Arguments

```
def main():  
    print('The sum of 12 and 45 is')  
    show_sum(12, 45)
```

```
def show_sum(num1, num2)  
    result = num1 + num2  
    print(result)
```



The diagram consists of two blue arrows. The first arrow originates from the number '12' in the function call `show_sum(12, 45)` within the `main` function and points down to the parameter `num1` in the function definition `def show_sum(num1, num2)`. The second arrow originates from the number '45' in the same function call and points down to the parameter `num2` in the function definition.

Functions

Returning Values

This function returns the square of a number:

```
def square(x):  
    return x*x
```

When Python encounters return, it exits the function and returns control to the point where the function was called.

In addition, the value(s) provided in the return statement are sent back to the caller as an expression result.

Functions

Returning Values

Sometimes a function needs to return more than one value.

To do this, simply list more than one expression in the return statement.

```
def sum_diff(x, y):  
    sum = x + y  
    diff = x - y  
    return sum, diff
```

Functions

Returning Values

When calling this function, use simultaneous assignment.

```
num1, num2 = eval(input("Enter two numbers (num1, num2) "))  
s, d = sum_diff(num1, num2)  
print("The sum is", s, "and the difference is", d)
```

As before, the values are assigned based on position, so *s* gets the first value returned (the sum), and *d* gets the second (the difference).

One “gotcha” – all Python functions return a value, whether they contain a return statement or not. Functions without a return hand back a special object, denoted `None`.

A common problem is writing a value-returning function and omitting the return!

Recursion

It is legal for one function to call another; it is also legal for a function to call itself.

It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do.

For example, look at the following function:

```
def countdown(n):  
    if n <= 0:  
        print('Blastoff!')  
    else:  
        print(n)  
        countdown(n-1)
```

Recursion

If n is 0 or negative, it outputs the word, “Blastoff!”

Otherwise, it outputs n and then calls a function named `countdown`—itself—passing $n-1$ as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```


Recursion

The execution of countdown begins with $n=3$, and since n is greater than 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with $n=2$, and since n is greater than 0, it outputs the value 2, and then calls itself...

The execution of countdown begins with $n=1$, and since n is greater than 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with $n=0$, and since n is not greater than 0, it outputs the word, "Blastoff!" and then returns.

The countdown that got $n=1$ returns.

The countdown that got $n=2$ returns.

Recursion

The countdown that got $n=3$ returns.

And then you're back in `__main__`. So, the total output looks like this:

3

2

1

Blastoff!

A function that calls itself is **recursive**; the process is called **recursion**.

Recursion

Stack diagrams

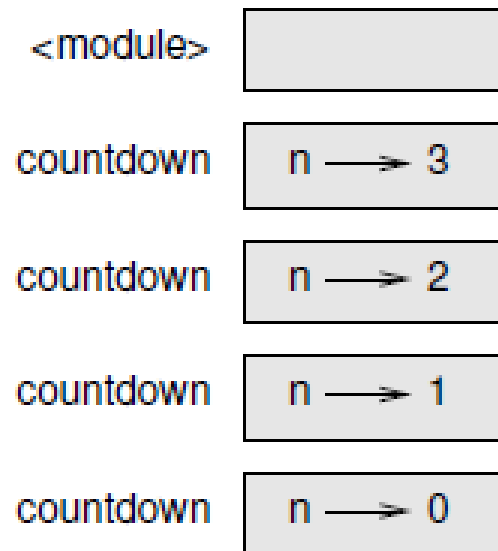


Fig: Stack Diagram

Infinite Recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates.

This is known as **infinite recursion**, and it is generally not a good idea.

Here is a minimal program with an infinite recursion:

```
def recurse():  
    recurse()
```

Infinite Recursion

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

Traceback (most recent call last):

File "C:\Users\pradeep\Desktop\bla.py", line 5, in <module>

recurse()

File "C:\Users\pradeep\Desktop\bla.py", line 3, in recurse

recurse()

File "C:\Users\pradeep\Desktop\bla.py", line 3, in recurse

recurse()

File "C:\Users\pradeep\Desktop\bla.py", line 3, in recurse

recurse()

[Previous line repeated 1022 more times]

RecursionError: maximum recursion depth exceeded

String

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

String representation

Inside the computer, strings are represented as sequences of 1's and 0's, just like numbers.

A string is stored as a sequence of binary numbers, one number per character.

It doesn't matter what value is assigned as long as it's done consistently.

In the early days of computers, each manufacturer used their own encoding of numbers for characters.

ASCII system (American Standard Code for Information Interchange) uses 127 bit codes.

Python supports Unicode (100,000+ characters).

String representation

The *ord* function returns the numeric (ordinal) code of a single character.

The *chr* function converts a numeric code to the corresponding character.

```
>>> ord("A")
```

```
65
```

```
>>> ord("a")
```

```
97
```

```
>>> chr(97)
```

```
'a'
```

```
>>> chr(65)
```

```
'A'
```


String

Using ord and char we can convert a string into and out of numeric form.

The encoding algorithm is simple:

- get the message to encode

- for each character in the message:

 - print the letter number of the character

A for loop iterates over a sequence of objects, so the for loop looks like:

- for ch in <string>

String

Example: Encoder

A program to convert a textual message into a sequence of
numbers, utilizing the underlying Unicode encoding.

```
def main():  
    print("This program converts a textual message into a sequence")  
    print("of numbers representing the Unicode encoding" + \  
          " of the message.\n")  
    message = input("Please enter the message to encode: ")  
  
    print("\nHere are the Unicode codes:")  
  
    for ch in message:  
        print(ord(ch), end=" ")  
    print() # blank line before prompt
```

String

Example: Decoder

We now have a program to convert messages into a type of “code”, but it would be nice to have a program that could decode the message!

The outline for a decoder:

- get the sequence of numbers to decode

- message = ""

- for each number in the input:

 - convert the number to the appropriate character

 - add the character to the end of the message

- print the message

String

Example: Decoder

A program to convert a sequence of Unicode numbers into a string of text.

```
def main():  
    print ("This program converts a sequence of Unicode numbers into")  
    print ("the string of text that it represents.\n")  
    s = input("Please enter the Unicode-encoded message: ")  
    message = ""  
  
    for num_str in s.split():  
        code_num = int(num_str)  
        message = message + chr(code_num)  
  
    print("\nThe decoded message is:", message)
```

String

Output of Encoder and Decoder

This program converts a textual message into a sequence of numbers representing the Unicode encoding of the message.

Please enter the message to encode: Python is fun

Here are the Unicode codes:

80 121 116 104 111 110 32 105 115 32 102 117 110

=====

This program converts a sequence of Unicode numbers into the string of text that it represents.

Please enter the Unicode-encoded message: 80 121 116 104 111 110 32 105 115 32 102 117 110

The decoded message is: Python is fun

String methods

There are a number of other string methods.

- `s.capitalize()` – Copy of `s` with only the first character capitalized
- `s.title()` – Copy of `s`; first character of each word capitalized
- `s.center(width)` – Center `s` in a field of given width
- `s.count(sub)` – Count the number of occurrences of `sub` in `s`
- `s.find(sub)` – Find the first position where `sub` occurs in `s`
- `s.join(list)` – Concatenate list of strings into one large string using `s` as separator.
- `s.ljust(width)` – Like center, but `s` is left-justified

String methods

- `s.lower()` – Copy of `s` in all lowercase letters
- `s.lstrip()` – Copy of `s` with leading whitespace removed
- `s.replace(oldsub, newsub)` – Replace occurrences of `oldsub` in `s` with `newsub`
- `s.rfind(sub)` – Like `find`, but returns the right-most position
- `s.rjust(width)` – Like `center`, but `s` is right-justified
- `s.rstrip()` – Copy of `s` with trailing whitespace removed
- `s.split()` – Split `s` into a list of substrings
- `s.upper()` – Copy of `s`; all characters converted to uppercase

Sequences

List

Sequence: an object that contains multiple items of data.

The items are stored in sequence one after another.

Python provides different types of sequences, including lists and tuples. The difference between these is that a list is mutable and a tuple is immutable.

List: an object that contains multiple data items

Element: An item in a list

Format: *list* = [*item1*, *item2*, etc.]

Can hold items of different types

print function can be used to display an entire list

list() function can convert certain types of objects to lists

List



A list of integers



A list of strings



A list holding different types

List

You can iterate over a list using a for loop

Format: for x in *list*:

Index: a number specifying the position of an element in a list.

- Enables access to individual element in list.
- Index of first element in the list is 0, second element is 1, and n th element is $n-1$.
- Negative indexes identify positions relative to the end of the list.
- The index -1 identifies the last element, -2 identifies the next to last element, etc.

List

An `IndexError` exception is raised if an invalid index is used.

len function: returns the length of a sequence such as a list

- Example: `size = len(my_list)`
- Returns the number of elements in the list, so the index of last element is `len(list) - 1`
- Can be used to prevent an `IndexError` exception when iterating over a list with a loop

Lists are mutable

Mutable sequence: the items in the sequence can be changed.

Lists are mutable, and so their elements can be changed.

An expression such as

`list[1] = new_value` can be used to assign a new value to a list element.

Must use a valid index to prevent raising of an `IndexError` exception.

Concatenating Lists

Concatenate: join two things together.

The + operator can be used to concatenate two lists.

Cannot concatenate a list with another data type, such as a number.

The += augmented assignment operator can also be used to concatenate lists

List Slicing

Slice: a span of items that are taken from a sequence.

- List slicing format: *list[start : end]*.
- Span is a list containing copies of elements from *start* up to, but not including, *end*
 - If *start* not specified, 0 is used for start index.
 - If *end* not specified, `len(list)` is used for end index.
- Slicing expressions can include a step value and negative indexes relative to end of list.

List methods

- **append(*item*)**: used to add items to a list – *item* is appended to the end of the existing list.
- **index(*item*)**: used to determine where an item is located in a list.
 - Returns the index of the first element in the list containing item.
 - Raises ValueError exception if *item* not in the list.
- **insert(*index*, *item*)**: used to insert *item* at position *index* in the list.

List methods

- **sort()**: used to sort the elements of the list in ascending order.
- **remove(*item*)**: removes the first occurrence of *item* in the list.
- **reverse()**: reverses the order of the elements in the list.
- **del statement**: removes an element from a specific index in a list
General format: `del list[i]`

List

Built-in functions

- **min and max functions:**

built-in functions that returns the item that has the lowest or highest value in a sequence

The sequence is passed as an argument

- **sum function:**

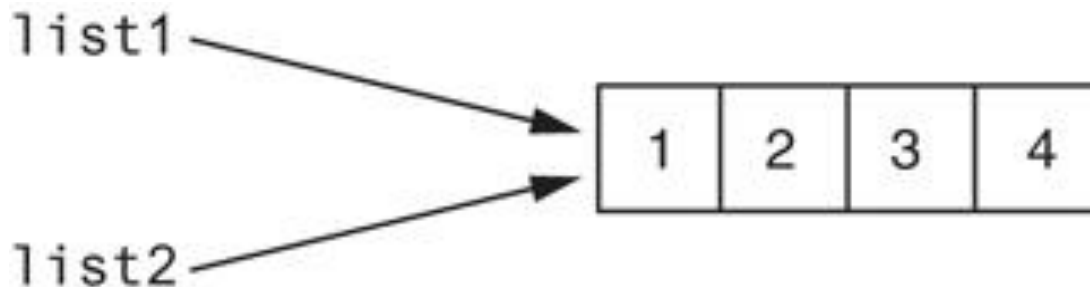
built-in function that returns the total of the numerical values in a sequence

Copying Lists

To make a copy of a list you must copy each element of the list.

Two methods to do this:

- Creating a new empty list and using a for loop to add a copy of each element from the original list to the new list.
- Creating a new empty list and concatenating the old list to the new empty list.



List Comprehensions

A concise expression that creates a new list by iterating over the elements of an existing list.

The following code uses a for loop to make a copy of a list:

```
list1 = [1, 2, 3, 4]
list2 = []

for item in list1:
    list2.append(item)
```

The following code uses a list comprehension to make a copy of a list:

```
list1 = [1, 2, 3, 4]
list2 = [item for item in list1]
```

List Comprehensions

```
list2 = [item for item in list1]
```

Result Expression Iteration Expression

- The iteration expression works like a for loop.
- In this example, it iterates over the elements of list1.
- Each time it iterates, the target variable item is assigned the value of an element.
- At the end of each iteration, the value of the result expression is appended to the new list.

List Comprehensions

```
list1 = [1, 2, 3, 4]  
list2 = [item**2 for item in list1]
```

After this code executes, list2 will contain the values [1, 4, 9, 16]

```
str_list = ['Winken', 'Blinken', 'Nod']  
len_list = [len(s) for s in str_list]
```

After this code executes, len_list will contain the values [6, 7, 3]

List Comprehensions

You can use an **if** clause in a list comprehension to select only certain elements when processing a list.

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = []
```

```
for n in list1:
    if n < 10:
        list2.append(n)
```

Works the same as:

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = [item for item in list1 if item < 10]
```

Two-dimensional Lists

- Two-dimensional list: a list that contains other lists as its elements.
 - Also known as nested list.
 - Common to think of two-dimensional lists as having rows and columns.
 - Useful for working with multiple sets of data
- To process data in a two-dimensional list need to use two indexes.
- Typically use nested loops to process.

Two-dimensional Lists

	Column 0	Column 1
Row 0	'Joe'	'Kim'
Row 1	'Sam'	'Sue'
Row 2	'Kelly'	'Chris'

	Column 0	Column 1	Column 2
Row 0	<code>scores[0][0]</code>	<code>scores[0][1]</code>	<code>scores[0][2]</code>
Row 1	<code>scores[1][0]</code>	<code>scores[1][1]</code>	<code>scores[1][2]</code>
Row 2	<code>scores[2][0]</code>	<code>scores[2][1]</code>	<code>scores[2][2]</code>

Tuples

Tuple: an immutable sequence.

- Very similar to a list.
- Once it is created it cannot be changed.
- Format: `tuple_name = (item1, item2)`.
- Tuples support operations as lists.
 - Subscript indexing for retrieving elements.
 - Methods such as `index`.
 - Built in functions such as `len`, `min`, `max`.
 - Slicing expressions.

Tuples

Advantages for using tuples over lists:

- Processing tuples is faster than processing lists.
- Tuples are safe.
- Some operations in Python require use of tuples.

list() function: converts tuple to list.

tuple() function: converts list to tuple.

Dictionaries

Dictionary: object that stores a collection of data.

- Each element consists of a **key** and a **value**.
 - Often referred to as **mapping** of key to value.
 - Key must be an immutable object.
- To retrieve a specific value, use the key associated with it.
- Format for creating a dictionary:

dictionary =

{key1:val1, key2:val2}

Retrieving a Value From a Dictionary

- Elements in dictionary are unsorted.
- General format for retrieving value from dictionary: *dictionary[key]*
 - If key in the dictionary, associated value is returned, otherwise, `KeyError` exception is raised.
- Test whether a key is in a dictionary using the `in` and `not in` operators.
 - Helps prevent `KeyError` exceptions.

Dictionary

Adding / Deleting element

Dictionaries are mutable objects.

To add a new key-value pair:

```
dictionary[key] = value
```

If key exists in the dictionary, the value associated with it will be changed.

To delete a key-value pair:

```
del dictionary[key]
```

If key is not in the dictionary, `KeyError` exception is raised.

Dictionary

len function: used to obtain number of elements in a dictionary.

Keys must be immutable objects, but associated values can be any type of object.

One dictionary can include keys of several different immutable types.

Values stored in a single dictionary can be of different types.

To create an empty dictionary:

- Use {}
- Use built-in function dict()
- Elements can be added to the dictionary as program executes

Use a for loop to iterate over a dictionary

General format: for *key* in *dictionary*:

Some dictionary methods

clear method: deletes all the elements in a dictionary, leaving it empty

Format: *dictionary.clear()*

get method: gets a value associated with specified key from the dictionary

- Format: *dictionary.get(key, default)*
default is returned if *key* is not found
- Alternative to `[]` operator
Cannot raise `KeyError` exception

Some dictionary methods

- **items method**: returns all the dictionaries keys and associated values.
Format: `dictionary.items()`
 - Returned as a **dictionary view**
Each element in dictionary view is a tuple which contains a key and its associated value.
 - Use a for loop to iterate over the tuples in the sequence.
Can use a variable which receives a tuple, or can use two variables which receive key and value.

Some dictionary methods

keys method: returns all the dictionaries keys as a sequence.

Format: `dictionary.keys()`

pop method: returns value associated with specified key and removes that key-value pair from the dictionary.

Format: `dictionary.pop(key, default)`

default is returned if key is not found.

values method: returns all the dictionaries values as a sequence

Format: `dictionary.values()`

Use a for loop to iterate over the values

Dictionary

Merge operator

The dictionary merge operator is the `|` symbol.

It merges two dictionaries into a single dictionary that is the combination of the two.

```
dict3 = dict1 | dict2
```

After this statement, `dict3` will have all the elements of `dict1` and `dict2`.

Dictionary

Update operator

The dictionary update operator is the `|=` symbol.

It works like the dictionary merge operator, but it assigns the new dictionary to the dictionary variable that appears on the left-hand side of the operator.

```
dict1 |= dict2
```

This statement merges `dict1` and `dict2`, and assigns the resulting dictionary back to `dict1`.

Sets

Set: object that stores a collection of data in same way as mathematical set.

- All items must be unique.
- Set is unordered.
- Elements can be of different data types.

Creating a set

set function: used to create a set.

- For empty set, call `set()`
- For non-empty set, call `set(argument)` where *argument* is an object that contains iterable elements.
e.g., *argument* can be a list, string, or tuple.

If *argument* is a string, each character becomes a set element.

For set of strings, pass them to the function as a list.

If *argument* contains duplicates, only one of the duplicates will appear in the set.

Some set methods

len function: returns the number of elements in the set

Sets are mutable objects.

add method: adds an element to a set.

update method: adds a group of elements to a set.

Argument must be a sequence containing iterable elements, and each of the elements is added to the set.

Some set methods

remove and discard methods: remove the specified item from the set.

- The item that should be removed is passed to both methods as an argument.
- Behave differently when the specified item is not found in the set.
 - remove method raises a `KeyError` exception.
 - discard method does not raise an exception.

clear method: clears all the elements of the set.

Some set methods

Union of two sets: a set that contains all the elements of both sets.

To find the union of two sets:

- Use the union method.

Format: `set1.union(set2)`

- Use the `|` operator.

Format: `set1 | set2`

- Both techniques return a new set which contains the union of both sets.

Some set methods

Intersection of two sets: a set that contains only the elements found in both sets.

To find the intersection of two sets:

- Use the intersection method.
Format: `set1.intersection(set2)`

- Use the & operator.
Format: `set1 & set2`

- Both techniques return a new set which contains the intersection of both sets.

Some set methods

Difference of two sets: a set that contains the elements that appear in the first set but do not appear in the second set.

To find the difference of two sets:

- Use the difference method.
Format: `set1.difference(set2)`
- Use the `–` operator.
Format: `set1 - set2`

Some set methods

Symmetric difference of two sets: a set that contains the elements that are not shared by the two sets.

To find the symmetric difference of two sets:

- Use the `symmetric_difference` method.
Format: `set1.symmetric_difference(set2)`
- Use the `^` operator.
Format: `set1 ^ set2`

Some set methods

Subsets

Set A is subset of set B if all the elements in set A are included in set B.

To determine whether set A is subset of set B.

- Use the `issubset` method
Format: `setA.issubset(setB)`

- Use the `<=` operator
Format: `setA <= setB`

Some set methods

Subsets

Set A is superset of set B if it contains all the elements of set B.

To determine whether set A is superset of set B

- Use the issuperset method
Format: `setA.issuperset(setB)`
- Use the `>=` operator
Format: `setA >= setB`

File Handling

For program to retain data between the times it is run, you must save the data.

- Data is saved to a file, typically on computer disk.
- Saved data can be retrieved and used at a later time.

“Writing data to”: saving data on a file.

Output file: a file that data is written to.

“Reading data from”: process of retrieving data from a file

Input file: a file from which data is read

- **Three steps when a program uses a file**
 - Open the file
 - Process the file
 - Close the file

Opening a file

open function: used to open a file

- Creates a file object and associates it with a file on the disk
- General format:
- `file_object = open(filename, mode)`

Mode: string specifying how the file will be opened

Example: reading only ('r'), writing ('w'), and appending ('a')

Writing data to a file

Method: a function that belongs to an object

Performs operations using that object

File object's write method used to write data to the file

Format: `file_variable.write(string)`

File should be closed using file object close method

Format: `file_variable.close()`

Reading data from a file

read method: file object method that reads entire file contents into memory

- Only works if file has been opened for reading
- Contents returned as a string

readline method: file object method that reads a line from the file

Line returned as a string, including '\n'.

Read position: marks the location of the next item to be read from a file

Concatenating a newline to and stripping it from a string

In most cases, data items written to a file are values referenced by variables

Usually necessary to concatenate a '\n' to data before writing it

Carried out using the + operator in the argument of the write method

In many cases need to remove '\n' from string after it is read from a file

rstrip method: string method that strips specific characters from end of the string

Appending data to an existing file

When open file with 'w' mode, if the file already exists it is overwritten

To append data to a file use the 'a' mode

- If file exists, it is not erased, and if it does not exist it is created
- Data is written to the file at the end of the current contents

Using loops to process files

Files typically used to hold large amounts of data

Loop typically involved in reading from and writing to a file

Often the number of items stored in file is unknown

The readline method uses an empty string as a sentinel when end of file is reached

Can write a while loop with the condition

`while line != ''`

Python allows the programmer to write a for loop that automatically reads lines in a file and stops when end of file is reached

Format: `for line in file_object:`

`statements`

The loop iterates once over each line in the file

Using with statement to open files

The with statement can be used to open a file and automatically close the file.

General format:

```
with open(filename, mode) as file_variable:  
    statement  
    statement  
    etc.
```

- filename is a string specifying the name of the file
- mode is a string specifying the mode, such as 'r', 'w', or 'a'
- file_variable is the name of the variable that will reference the file object
- The indented block of statements is known as the with suite

Using with statement to open files

Example:

```
with open('myfile.txt', 'r') as input_file:  
    statement  
    statement  
    etc.
```

- This example opens a file named myfile.txt.
- Code inside the with suite can use input_file to read data from the file.
- When the code inside the with suite has finished, the file is automatically closed.

Using with statement to open files

Opening multiple files with a with statement:

```
with open('file1.txt', 'r') as input_file,  
     open('file2.txt', 'w') as output_file:  
    statement  
    statement  
    etc.
```

- This example opens a file named file1.txt for reading, and a file named file2.txt for writing
- Code inside the with suite can use input_file to read data from file1.txt and output_file to write data to file2.txt
- When the code inside the with suite has finished, both files are automatically closed