18CSC207J – Advanced Programming Practice

Course Content

Unit 1
- Structured Programming Paradigm
- Procedural Programming Paradigm
- Object Oriented Programming Paradigm

Unit 2
- Event Driven Programming Paradigm
- Declarative Programming Paradigm
- Imperative Programming Paradigm

Unit 3
- Parallel Programming Paradigm
- Concurrent Programming paradigm
- Functional Programming Paradigm

Unit 4
- Logic Programming Paradigm
- Dependent Type Programming Paradigm
- Network Programming Paradigm

Unit 5
- Symbolic Programming paradigm
- Automata Programming Paradigm
- GUI Programming paradigm

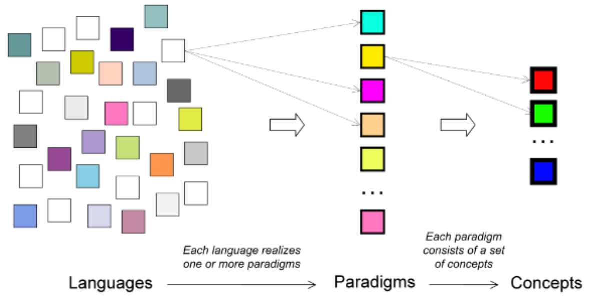
Introduction to Programming Paradigm

Introduction to Programming Paradigm

Hundreds of programming languages are in use...

Introduction to Programming Paradigm

So many, how can we understand them all?


Each language realizes one or more paradigms
Each paradigm consists of a set of concepts
Languages
Paradigms
Concepts

Key insight:

Languages are based on paradigms, and there are many fewer paradigms than languages

We can ***understand many languages by learning few paradigms***

What is a Programming Paradigm?

- A programming paradigm is an approach to programming a computer based on a coherent set of principles or a mathematical theory
- A program is written to solve problems
  - Any realistic program needs to solve different kinds of problems
  - Each kind of problem needs its own paradigm
  - So we need multiple paradigms and we need to combine them in the same program

How can we study multiple paradigms? How can we combine paradigms in a program?

How can we study multiple paradigms without studying multiple languages (since most languages only support one, or sometimes two paradigms)?

- Each language has its own syntax, its own semantics, its own system, and its own quirks
  - We could pick python language and structure our course around them
- Each paradigm is a different way of thinking
  - How can we combine different ways of thinking in one program?
- We can do it using the concept of a kernel language
  - Each paradigm has a simple core language, its kernel language, that contains its essential concepts
    - Every practical language, even if it's complicated, can be translated easily into its kernel language
  - Even very different paradigms have kernel languages that have much in common; often there is only one concept difference
- We start with a simple kernel language that underlies our first paradigm, functional programming
  - We then add concepts one by one to give the other paradigms

Introduction to Programming Languages

- A ***programming language is an artificial language*** designed to communicate
  - instructions to a machine, e.g., computer
- The earliest programming languages preceded the invention of the computer
  - e.g., used to direct the behavior of machines such as Jacquard looms and player pianos.
- *"Programming languages are the **least usable, but most powerful human-computer interfaces** ever invented"*

Introduction to Programming Languages

- Programming languages provide an ***abstraction*** from a computer's instruction set architecture
- ***Low-level programming languages*** provide little or no abstraction, e.g., machine code and assembly language
  - Difficult to use
  - Allows to program efficiently and with a low memory footprint
- ***High-level programming languages*** isolate the execution semantics of a computer architecture from the specification of the program
  - Simplifies program development

Machine code

```
8B542408 83FA0077 06B80000 0000C383
C9010000 008D0419 83FA0376 078BD98B
B84AEBF1 5BC3
```

Assembly language

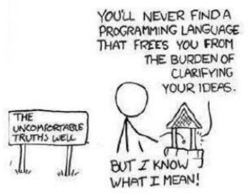
```
mov edx, [esp+8]
cmp edx, 0
ja @f
mov eax, 0
ret
```

High-level language

```
unsigned int fib(unsigned int n) {
if (n <= 0)
return 0;
else if (n <= 2)
return 1;
else
...
}
```

Introduction to Programming Languages

- Programming languages can be categorized into ***programming paradigms***
- Meaning of the word ***'paradigm'***
  - "An example that serves as pattern or model"
  - "***Paradigms*** emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas"
- Programming paradigms are the result of people's ideas about how computer programs should be constructed
  - Patterns that serves as a "***school of thoughts***" for programming of computers
- Once you have understood the general concepts of programming paradigms, it becomes easier to learn new programming languages


YOU'LL NEVER FIND A PROGRAMMING LANGUAGE THAT FREES YOU FROM THE BURDEN OF CLARIFYING YOUR IDEAS.
THE UNCOMFORTABLE TRUTHS WELL
BUT I KNOW WHAT I MEAN!

Basics of Programming Language

Elements of Programming Language

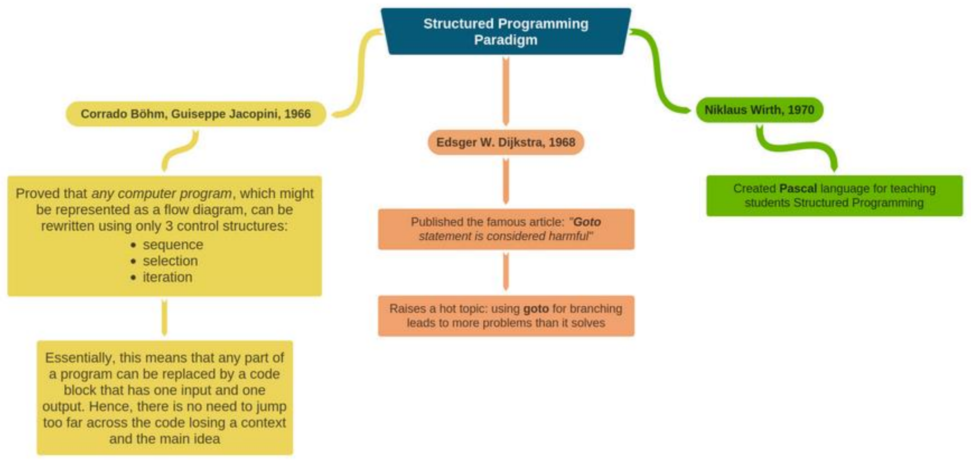
- Programming languages have many similarities with natural languages
  - e.g., they conform to rules for syntax and semantics, there are many dialects, etc.
- We are going to have a quick look at the following concepts
  - Compiled/Interpreted
  - Syntax
  - Semantics
  - Typing

Structured Programming Paradigm

1. Structured Programming Paradigm

- Program is made as a single structure.
- Code will execute the instruction by instruction one after the other.
- It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc.
- The structured program consists of well structured and separated modules. But the entry and exit in a Structured program is a single-time event. It means that the program uses single-entry and single-exit elements.
- Instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are: C, C++, Java, C# ..etc
- The structured program mainly consists of three types of elements:
  - Selection Statements
  - Sequence Statements
  - Iteration Statements

1. Structured Programming Paradigm


Structured Programming Paradigm

Corrado Böhm, Guiseppe Jacopini, 1966

Proved that *any computer program*, which might be represented as a flow diagram, can be rewritten using only 3 control structures:
- sequence
- selection
- iteration

Essentially, this means that any part of a program can be replaced by a code block that has one input and one output. Hence, there is no need to jump too far across the code losing a context and the main idea

Edsger W. Dijkstra, 1968

Published the famous article: "*Goto statement is considered harmful*"

Raises a hot topic: using **goto** for branching leads to more problems than it solves

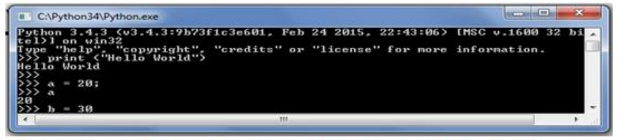
Niklaus Wirth, 1970

Created **Pascal** language for teaching students Structured Programming

1. Structured Programming Paradigm

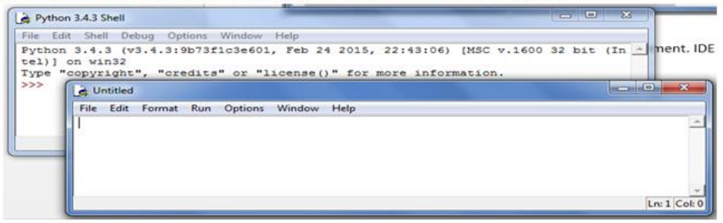
Starting Python

- Python is invoked in two different mode
  - Interactive Mode (By invoking python Interpreter from command prompt)



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bi
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello World")
Hello World
>>>
>>> a = 20;
>>> a
20
>>> b = 30
```

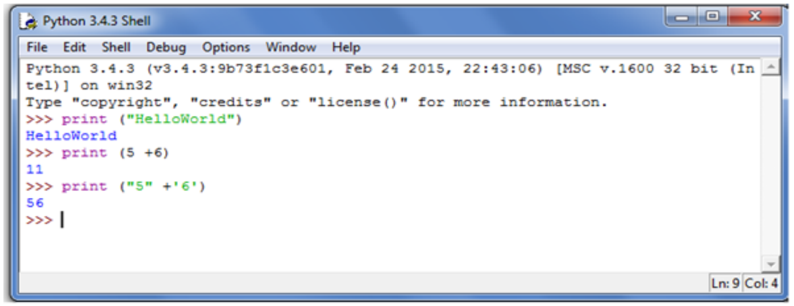
  - Script Mode (By using IDLE –Interactive Development Environment. IDE used for python)


Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Untitled
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

1. Structured Programming Paradigm

Working with command prompt



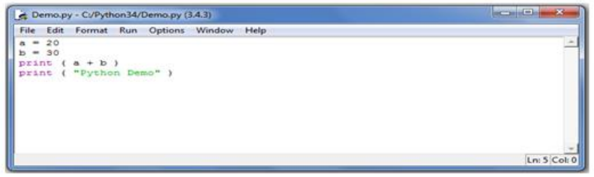
```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print ("HelloWorld")
HelloWorld
>>> print (5 +6)
11
>>> print ("5" +'6')
56
>>> |
```

- Print is the built in function used to print the result back to the console. Console refers to text entry (keyboard) or display device (monitor) of the computer

1. Structured Programming Paradigm

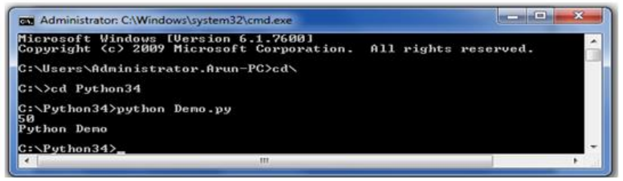
Creating python source file

- Python source file is created using IDLE or notepad and save the contents with .py extension. It reference python source file, script file or module.


Demo.py - C:/Python34/Demo.py (3.4.3)
File Edit Format Run Options Window Help
Ln: 5 Col: 0

```
a = 20
b = 30
print ( a + b )
print ( "Python Demo" )
```

- Execute the python file using IDLE or execute from command prompt



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Administrator.Arun-PC>cd\

C:\>cd Python34

C:\Python34>python Demo.py
50
Python Demo

C:\Python34>_
```

Selection Statement

If else
Syntax:

```
if test expression:
        statement(s)
```

If else
Syntax:

```
if test expression:
      Body of if
else:
      Body of else
```

If elif else
Syntax:

```
if test expression:
   Body of if
elif test expression:
   Body of elif
else:
   Body of else
```

Example:

```
num = float(input("Enter a number: "))
if num > 0:
   print("Positive number")
print("This is always printed")
```

Example:

```
num = float(input("Enter a number: "))
if num >= 0:
             print("Positive or Zero")
else:
             print("Negative number")
```

Example:

```
num = float(input("Enter a number: "))
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

1. Structured Programming Paradigm

Conditional

•A conditional expression evaluates an expression based on a condition.

•Conditional expression is expressed using **if** and **else** combined with expression

Syntax:

expression if Boolean-expression else expression

Example:

Biggest of two numbers

```
num1 = 23
num2 = 15
big = num1 if num1 > num2 else num2
print ( " the biggest number is " , big )
```

Even or odd

```
print ( " num is even " if num % 2 == 0 else " num is odd ")
```

1. Structured Programming Paradigm

Looping

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for or do..while loop.

1. Structured Programming Paradigm

while loop

Syntax:
```
        while expression:
          statement(s)
```

Example:
```
count = 0
while (count < 9):
  print 'The count is:', count
  count = count + 1
print "Good bye!"
```

For Loop

Syntax:
```
        for iterating_var in sequence:
          statements(s)
```

Example:
```
for letter in 'Python':     # First Example
        print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
        print 'Current fruit :',
fruits[index]
print "Good bye!“
```

Else with looping:

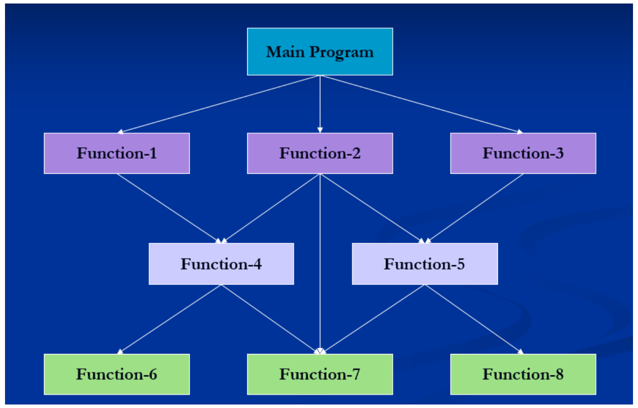
Example
```
for num in range(10,20):  #to iterate between 10 to 20
  for i in range(2,num): #to iterate on the factors of the number
    if num%i == 0:      #to determine the first factor
      j=num/i         #to calculate the second factor
      print '%d equals %d * %d' % (num,i,j)
      break #to move to the next number, the #first FOR
  else:                # else part of the loop
    print num, 'is a prime number'
```

Procedure Programming Paradigm

Introduction

- High level languages such as COBOL, FORTRAN and C, is commonly known as procedure oriented programming(POP). In the procedure oriented programming, program is divided into sub programs or modules and then assembled to form a complete program. These modules are called functions.
- The problem is viewed as a sequence of things to be done.
- The primary focus is on functions.
- Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow and organizing these instructions into groups known as functions.
- In a multi-function program, many important data items are placed as global so that they may be accessed by all functions. Each function may have its own local data. If a function made any changes to global data, these changes will reflect in other functions. Global data are more unsafe to an accidental change by a function. In a large program it is very difficult to identify what data is used by which function.
- This approach does not model real world problems. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Typical structure of procedure-oriented program



Characteristics of Procedure-Oriented Programming

- Emphasis is on doing things.
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Logical view and Control flow of POP (routine, subroutine and function)

- Procedural programming is a programming paradigm, derived from structured programming, based on the concept of the procedure call. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.
- procedural languages generally use reserved words that act on blocks, such as if, while, and for, to implement control flow, whereas non-structured imperative languages use goto statements and branch tables for the same purpose.

Note:

- Subroutine:-
- Subroutines; callable units such as procedures, functions, methods, or subprograms are used to allow a sequence to be referred to by a single statement.

Functions in Python

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.
- There are 2 types of function
-     Built-in function   ex. Print()
-     User defined function -User can create their own functions.

Defining a Function

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
def functionname( parameters ):
     "function_docstring"
          function_suite
  return [expression]
```

Function Arguments

- You can call a function by using the following types of formal arguments –
- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Function Arguments

We want to make some of its parameters as optional and use default values if the user does not want to provide values for such parameters.

Example:

```
def say(message, times = 1):
        print message * times

say('Hello')
say('World', 5)
```

Note : Default parameters placed at the end of the parameter list.

Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called keyword arguments. By specifying the name of the parameter we can substitute the value.

Advantages

one, using the function is easier since we do not need to worry about the or-der of the arguments.

we can give values to only those parameters which we want, provided that the other parameters have default argument values.

Example:

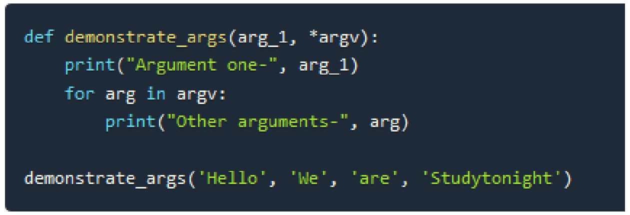
```
def func(a, b=5, c=10):
        print 'a is', a, 'and b is', b, 'and c is', c
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Variable length Parameter

- Python allows us to create functions that can take multiple arguments. So, lets create multi-argument functions.
- In Python, by adding * and ** (one or two asterisks) to the head of parameter names in the function definition, you can specify an arbitrary number of arguments (variable-length arguments) when calling the function.
- By convention, the names *args (arguments) and **kwargs (keyword arguments) are often used, but as long as * and ** are headed, there are no problems with other names. The sample code below uses the names *args and **kwargs.

Example:

```
def my_sum(*args):
  return sum(args)

print(my_sum(1, 2, 3, 4))                # 10
print(my_sum(1, 2, 3, 4, 5, 6, 7, 8))    # 36
```



```
def demonstrate_args(arg_1, *argv):
    print("Argument one-", arg_1)
    for arg in argv:
        print("Other arguments-", arg)

demonstrate_args('Hello', 'We', 'are', 'Studytonight')
```

Anonymous function

Usually in any programming function declaration should have a valid identifier as name to be invoked, but python supports user defined function to be defined without any name. Such function are called as anonymous function or lambda function.

Syntax:

```
lambda arguments: expression
```

Example:

```
double = lambda x: x * 2
print(double(5))
```

Lambda functions are used along with built-in functions like filter(), map() etc

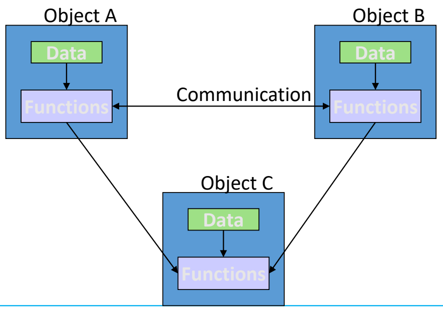
Example:

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
```

Object Oriented Programming Paradigm

Introduction

- OOP treat data as a critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.
- OOP allows decomposition of a problem into a number of entities called objects and then build data functions around these objects.
- The data of an object can be accessed only by the functions associated with that object.
- Functions of one object can access the functions of another objects



Characteristics

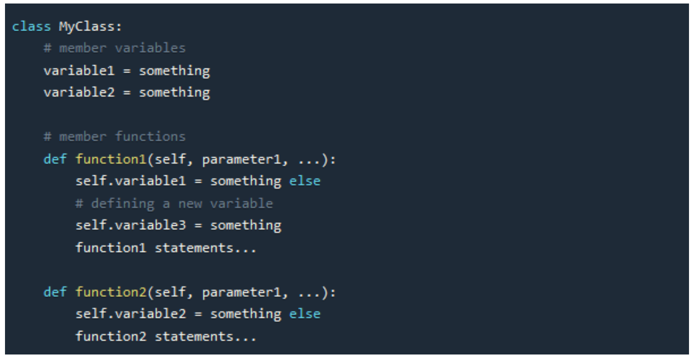
- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and can not be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be added easily whenever necessary.
- Follows bottom-up approach in program design.

Basic Concepts of Object-Oriented Programming

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

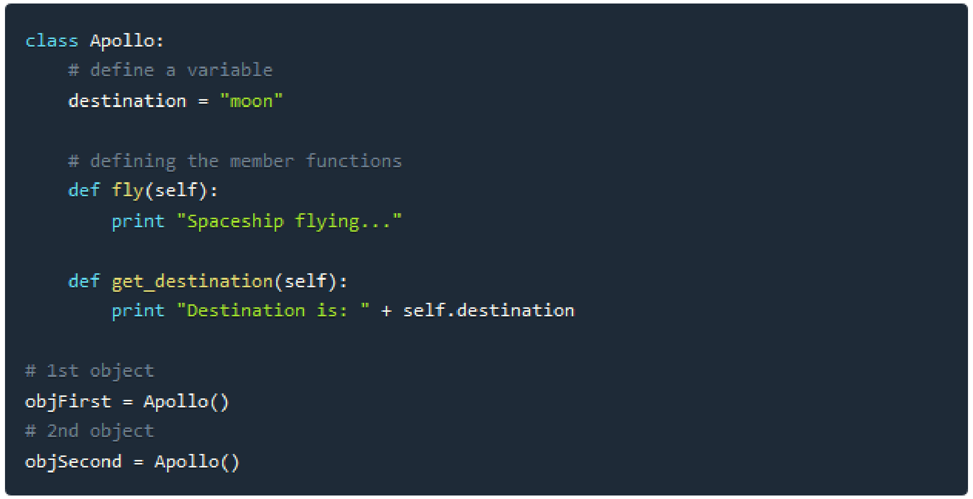
Classes in Python

- Defining a class is simple, all you have to do is use the keyword class followed by the name that you want to give your class, and then a colon symbol :. It is standard approach to start the name of class with a capital letter and then follow the camel case style.
- The class definition is included, starting from the next line and it should be indented, as shown in the code below. Also, a class can have variables and member functions in it.



```
class MyClass:
    # member variables
    variable1 = something
    variable2 = something

    # member functions
    def function1(self, parameter1, ...):
        self.variable1 = something else
        # defining a new variable
        self.variable3 = something
        function1 statements...

    def function2(self, parameter1, ...):
        self.variable2 = something else
        function2 statements...
```

Classes in Python

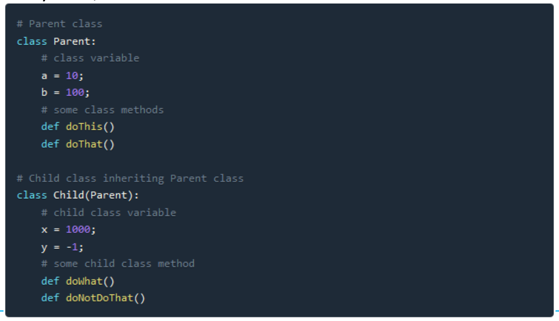


```
class Apollo:
    # define a variable
    destination = "moon"

    # defining the member functions
    def fly(self):
        print "Spaceship flying..."

    def get_destination(self):
        print "Destination is: " + self.destination

# 1st object
objFirst = Apollo()
# 2nd object
objSecond = Apollo()
```

Classes

Note:

- We add the self parameter when we define a member function, but do not specify it while calling the function.
- When we called get_destination function for objFirst it gave output as Destination is: mars, because we updated the value for the variable destination for the object objFirst
- To access a member function or a member variable using an object, we use a dot . symbol.
- And to create an object of any class, we have to call the function with same name as of the class.



```
# Parent class
class Parent:
    # class variable
    a = 10;
    b = 100;
    # some class methods
    def doThis()
    def doThat()

# Child class inheriting Parent class
class Child(Parent):
    # child class variable
    x = 1000;
    y = -1;
    # some child class method
    def doWhat()
    def doNotDoThat()
```

Constructor and Destructor

```
class Example:
    def __init__(self):
        print "Object created"

    # destructor
    def __del__(self):
        print "Object destroyed"

# creating an object
myObj = Example()
# to delete the object explicitly
del myObj
```

Note:

Like Destructor is counter-part of a Constructor, function __del__ is the counter-part of function __new__. Because __new__ is the function which creates the object.

__del__ method is called for any object when the reference count for that object becomes zero.

As reference counting is performed, hence it is not necessary that for an object __del__ method will be called if it goes out of scope. The destructor method will only be called when the reference count becomes zero.

Function Overloading

- In OOP, it is possible to make a function act differently using function overloading. All we have to do is, create different functions with same name having different parameters. For example, consider a function add(), which adds all its parameters and returns the result. In python we will define it as,

```
def add(a, b):
    return a + b
```

- Python doesn't support method overloading on the basis of different number of parameters in functions.

```
# to add 3 numbers
def add(a, b, c):
    return a + b + c

# to add 4 numbers
def add(a, b, c, d):
    return a + b + c + d
```

Function Overloading

def add(a,b):

        return a+b

def add(a,b,c):

        return a+b+c

print add(4,5)

- If you try to run the above piece of code, you get an error stating, “TypeError: add() takes exactly 3 arguments (2 given)”. This is because, Python understands the latest definition of method add() which takes only two arguments. Even though a method add() that takes care of three arguments exists, it didn’t get called. Hence you would be safe to say, overloading methods in Python is not supported.

Inheritance

```
# Parent class
class Parent:
    # class variable
    a = 10;
    b = 100;
    # some class methods
    def doThis()
    def doThat()

# Child class inheriting Parent class
class Child(Parent):
    # child class variable
    x = 1000;
    y = -1;
    # some child class method
    def doWhat()
    def doNotDoThat()
```

Method overriding

Method overriding is a concept of object oriented programming that allows us to change the implementation of a function in the child class that is defined in the parent class. It is the ability of a child class to change the implementation of any method which is already provided by one of its parent class(ancestors).

Following conditions must be met for overriding a function:

Inheritance should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.

The function that is redefined in the child class should have the same signature as in the parent class i.e. same number of parameters.

Method Overriding

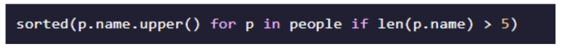
```
# parent class
class Animal:
 # properties
          multicellular = True
          # Eukaryotic means Cells with Nucleus
          eukaryotic = True
          # function breath
          def breathe(self):
            print("I breathe oxygen.")
   # function feed
          def feed(self):
            print("I eat food.")
# child class
class Herbivorous(Animal):
 # function feed
          def feed(self):
               print("I eat only plants. I am vegetarian.")

herbi = Herbivorous()
herbi.feed()
# calling some other function
herbi.breathe()
```

Functional Programming Paradigm

Introduction

- Functional programming is a programming paradigm in which it is tried to bind each and everything in pure mathematical functions. It is a declarative type of programming style that focuses on what to solve rather than how to solve.
- Functional programming paradigm is based on lambda calculus.
- Instead of statements, functional programming makes use of expressions. Unlike a statement, which is executed to assign variables, evaluation of an expression produces a value.
- Functional programming is a declarative paradigm because it relies on expressions and declarations rather than statements. Unlike procedures that depend on a local or global state, value outputs in FP depend only on the arguments passed to the function.
- Functional programming consists only of PURE functions.
- In functional programming, control flow is expressed by combining function calls, rather than by assigning values to variables.

Example:

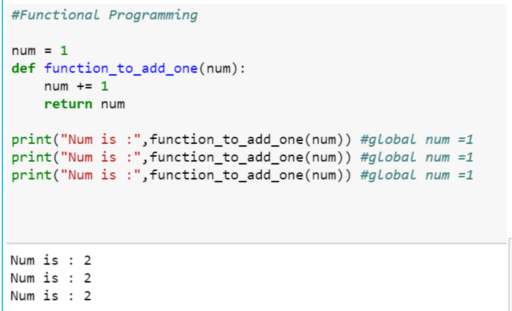


```
sorted(p.name.upper() for p in people if len(p.name) > 5)
```

Characteristics of Functional Programming

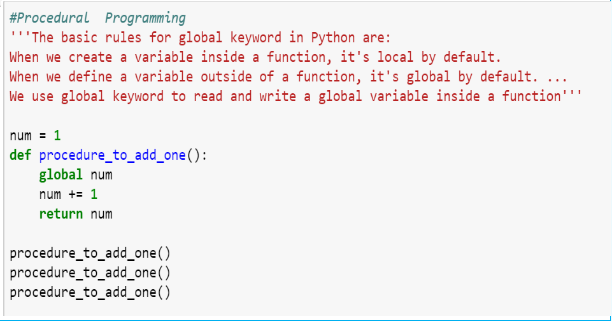
- Functional programming method focuses on results, not the process
- Emphasis is on what is to be computed
- Data is immutable
- Functional programming Decompose the problem into 'functions
- It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation
- It does not support iteration like loop statements and conditional statements like If-Else
- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports higher-order functions and lazy evaluation features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.

Functional Programming vs Procedure Programming



```
#Functional Programming

num = 1
def function_to_add_one(num):
    num += 1
    return num

print("Num is :",function_to_add_one(num)) #global num =1
print("Num is :",function_to_add_one(num)) #global num =1
print("Num is :",function_to_add_one(num)) #global num =1
```

```
Num is : 2
Num is : 2
Num is : 2
```



```
#Procedural  Programming
'''The basic rules for global keyword in Python are:
When we create a variable inside a function, it's local by default.
When we define a variable outside of a function, it's global by default. ...
We use global keyword to read and write a global variable inside a function'''

num = 1
def procedure_to_add_one():
    global num
    num += 1
    return num

procedure_to_add_one()
procedure_to_add_one()
procedure_to_add_one()
```

Mathematical Notation of Functional Programming

Notation:

F(x) = y

x , y -> Arguments or parameters

x -> domain and

y->codomain

Types:

1. Injective if f(a) ≠ f(b)
2. Surjective if f(X)= Y
3. Bijective (support both)

Functional Rules:

1. (f+g)(x)=f(x)+g(x)
2. (f-g)(x)=f(x)-g(x)
3. (f*g)(x)=f(x)*g(x)
4. (f/g)(x)=f(x)/g(x)
5. (gof)(x)=g(f(x))
6. f f-1=idy

Mathematical Notation of Functional Programming

```
f(x) = 3x 2 - 2x + 5
def f(x):
          return 3 * x ** 2 - 2 * x + 5
>>> f(3)
26
>>> f(0)
 5
>>> f(-1)
 10
>>> result = f(3) + f(-1)
>>> result  #output 36
```

Function composition is a way of combining functions such that the result of each function is passed as the argument of the next function. Example:

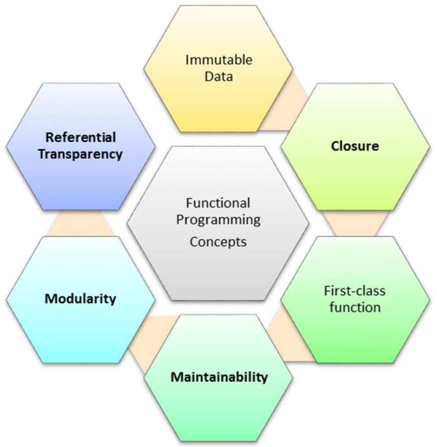
```
def f(x):
...    return 2 * x
>>> def g(x):
...    return x + 5
>>> def h(x):
...    return x ** 2 - 3
>>> f(3)            # 6
>>> g(3)            # 8
>>> h(4)            # 13
>>> f(g(3))         # 16
>>> g(f(3))         # 11
>>> h(f(g(0)))      # 97
```

Mathematical Notation of Functional Programming

```
def compose2(f, g):
        return lambda x: f(g(x))

>>> def double(x):
...     return x * 2

>>> def inc(x):
...     return x + 1
...
>>> inc_and_double = compose2(double, inc)
>>> inc_and_double(10)
#output 22
```

Concepts of FP

- Pure functions
- Recursion
- Referential transparency
- Functions are First-Class and can be Higher-Order
- Immutability

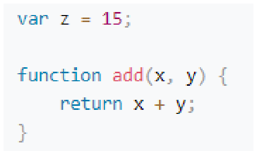


1. Pure functions

- Pure functions always return the same results when given the same inputs. Consequently, they have no side effects.
- Properties of Pure functions are:
  - First, they always produce the same output for same arguments irrespective of anything else.
  - Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.
- A simple example would be a function to receive a collection of numbers and expect it to increment each element of this collection.
- We receive the numbers collection, use map with the ***inc*** function to increment each number, and return a new list of incremented numbers.

Example:

```
def inc(x):
        return x+1
list=[8,3,7,5,2,6]
x=map(inc,list)   #print(list)
print(x)
```



```
var z = 15;

function add(x, y) {
    return x + y;
}
```

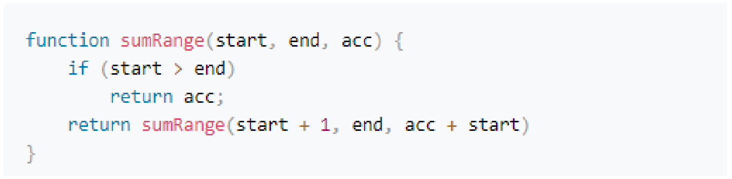
Note : Function involving Reading files,
using global data, random numbers are impure functions

Note: if a function relies on the global variable or class member's data, then it is not pure. And in such cases, the return value of that function is not entirely dependent on the list of arguments received as input and can also have side effects.

A side effect is a change in the state of an application that is observable outside the called function other than its return value

2. Recursion

- In the functional programming paradigm, there is no for and while loops. Instead, functional programming languages rely on recursion for iteration. Recursion is implemented using recursive functions, which repetitively call themselves until the base case is reached.

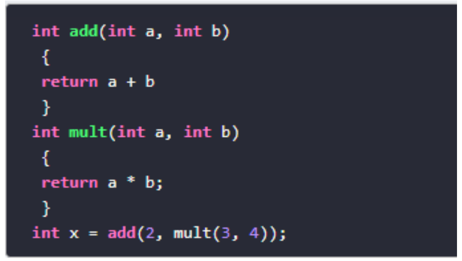


```
function sumRange(start, end, acc) {
    if (start > end)
        return acc;
    return sumRange(start + 1, end, acc + start)
}
```

- The above code performs recursion task as the loop by calling itself with a new start and a new accumulator.

Referential transparency

- An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behaviour. As a result, evaluating a referentially transparent function gives the same value for fixed arguments. If a function consistently yields the same result for the same input, it is referentially transparent.
- Functional programs don't have any assignment statements. For storing additional values in a program developed using functional programming, new variables must be defined. State of a variable in such a program is constant at any moment in time
    - pure function + immutable date = referential transparency

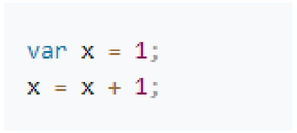


```
int add(int a, int b)
 {
 return a + b
 }
int mult(int a, int b)
 {
 return a * b;
 }
int x = add(2, mult(3, 4));
```

- Let's implement a square function:
    - This (pure) function will always have the same output, given the same input.
    - Passing "2" as a parameter of the square function will always returns 4. So now we can replace the (square 2) with 4.

Immutability

- In functional programming you cannot modify a variable after it has been initialized. You can create new variables and this helps to maintain state throughout the runtime of a program.



```
var x = 1;
x = x + 1;
```

- In imperative programming, this means "take the current value of x, add 1 and put the result back into x." In functional programming, however, x = x + 1 is illegal. That's because there are technically no variables in functional programming.
- Using immutable data structures, you can make single or multi-valued changes by copying the variables and calculating new values,
- Since FP doesn't depend on shared states, all data in functional code must be immutable, or incapable of changing

Functions are First-Class and can be Higher-Order

- A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.
- Higher-order functions are functions that take at least one first-class function as a parameter
- Examples:
    - name_lengths = map(len, ["Bob", "Rob", "Bobby"])
- Higher Order functions are map, reduce, filter

Functional style of getting a sum of a list:

```
new_lst = [1, 2, 3, 4]
def sum_list(lst):
   if len(lst) == 1:
            return lst[0]
   else:
            return lst[0] + sum_list(lst[1:])
print(sum_list(new_lst))
```

# or the pure functional way in python using higher order function

```
import functools
print(functools.reduce(lambda x, y: x + y, new_lst))
```

Functions are First-Class and can be Higher-Order

Map

map() can run the function on each item and insert the return values into the new collection. Example:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])
import random
names = ['Seth', 'Ann', 'Morganna']
team_names = map(lambda x: random.choice(['A Team','B Team']),names)
print names
// ['A Team', 'B Team', 'B Team']
```

reduce()

- reduce() is another higher order function for performing iterations. It takes functions and collections of items, and then it returns the value of combining the items

Example:

```
sum = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print sum          // 10
```

Functions are First-Class and can be Higher-Order ..cont

filter()

- filter function expects a true or false value to determine if the element should or should not be included in the result collection. Basically, if the call-back expression is true, the filter function will include the element in the result collection. Otherwise, it will not.

Example:

```
def f(x):
        return x%2 != 0 and x%3 ==0
filter(f, range(2,25))
```

Functional Programming – Non Strict Evaluation

- In programming language theory, lazy evaluation, or call-by-need[1] is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations
- Allows Functions having variables that have not yet been computed

In Python, the logical expression operators and, or, and if-then-else are all non-strict. We sometimes call them short-circuit operators because they don't need to evaluate all arguments to determine the resulting value.

The following command snippet shows the and operator's non-strict feature:

```
>>> 0 and print("right")
0
>>> True and print("right")
Right
```

When we execute the preceding command snippet, the left-hand side of the and operator is equivalent to False; the right-hand side is not evaluated. When the left-hand side is equivalent to True, the right-hand side is evaluated

Functional Programming – lambda calculus

Lambda expressions in Python and other programming languages have their roots in lambda calculus. Lambda calculus can encode any computation. Functional languages get their origin in mathematical logic and lambda calculus

In Python, we use the lambda keyword to declare an anonymous function, which is why we refer to them as "lambda functions". An anonymous function refers to a function declared with no name.

when you want to pass a function as an argument to higher-order functions, that is, functions that take other functions as their arguments

Characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

Syntax:

```
lambda argument(s): expression
```

Example:

```
remainder = lambda num: num % 2                    [(lambda x: x*x)(x) for x in [2,6,9,3,6,4,8]]
print(remainder(5))
```

Functional Programming – Closure

Basically, the method of binding data to a function without actually passing them as parameters is called closure. It is a function object that remembers values in enclosing scopes even if they are not present in memory.

Example:.

```
def counter(start=0, step=1):
    x = [start]
    def _inc():
        x[0] += step
        return x[0]
    return _inc
c1 = counter()
c2 = counter(100, -10)
c1()
//1
c2()
90
```

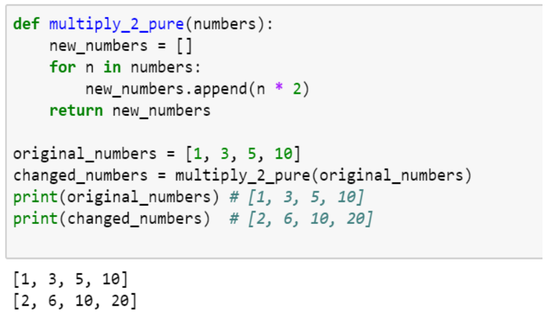
Functional Programming in Python

- Functional Programming is a popular programming paradigm closely linked to computer science's mathematical foundations. While there is no strict definition of what constitutes a functional language, we consider them to be languages that use functions to transform data.

- Python is not a functional programming language but it does incorporate some of its concepts alongside other programming paradigms. With Python, it's easy to write code in a functional style, which may provide the best solution for the task at hand.

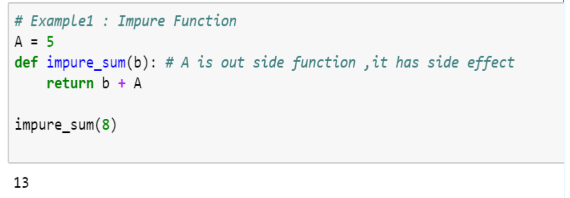
Pure Functions in Python

- If a function uses an object from a higher scope or random numbers, communicates with files and so on, it might be impure



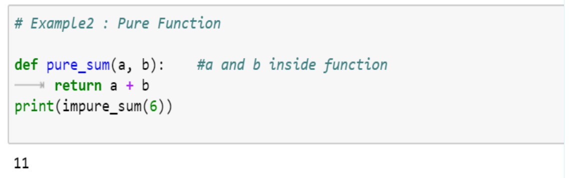
```
def multiply_2_pure(numbers):
    new_numbers = []
    for n in numbers:
        new_numbers.append(n * 2)
    return new_numbers

original_numbers = [1, 3, 5, 10]
changed_numbers = multiply_2_pure(original_numbers)
print(original_numbers) # [1, 3, 5, 10]
print(changed_numbers)  # [2, 6, 10, 20]
```

```
[1, 3, 5, 10]
[2, 6, 10, 20]
```



```
# Example1 : Impure Function
A = 5
def impure_sum(b): # A is out side function ,it has side effect
    return b + A

impure_sum(8)
```

```
13
```



```
# Example2 : Pure Function

def pure_sum(a, b):    #a and b inside function
    return a + b
print(impure_sum(6))
```

```
11
```

Built-in Higher Order Functions

Map

- The map function allows us to apply a function to every element in an iterable object

Filter

- The filter function tests every element in an iterable object with a function that returns either True or False, only keeping those which evaluates to True.

Combining map and filter

- As each function returns an iterator, and they both accept iterable objects, we can use them together for some really expressive data manipulations!

List Comprehensions

- A popular Python feature that appears prominently in Functional Programming Languages is list comprehensions. Like the map and filter functions, list comprehensions allow us to modify data in a concise, expressive way.

Anonymous Function

- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

Syntax of Lambda Function in python

lambda arguments: expression

Example:

```
double = lambda x: x * 2
print(double(5))
# Output: 10
```

```
product = lambda x, y : x * y
print(product(2, 3))
```

Note: you want to pass a function as an argument to higher-order functions, that is, functions that take other functions as their arguments

map() Function

Example Map with lambda

tup= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)

newtuple = tuple(map(lambda x: x+3 , tup))

print(newtuple)

//with multiple iterables

list_a = [1, 2, 3]

list_b = [10, 20, 30]

map(lambda x, y: x + y, list_a, list_b)

Example with Map

from math import sqrt

map(sqrt, [1, 4, 9, 16])

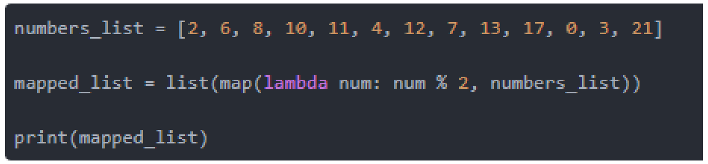
[1.0, 2.0, 3.0, 4.0]

map(str.lower, ['A', 'b', 'C'])

['a', 'b', 'c']

#splitting the input and convert to int using map

print(list(map(int, input.split(‘ ‘))))



```
numbers_list = [2, 6, 8, 10, 11, 4, 12, 7, 13, 17, 0, 3, 21]

mapped_list = list(map(lambda num: num % 2, numbers_list))

print(mapped_list)
```

map() Function

- map() function is a type of higher-order. As mentioned earlier, this function takes another function as a parameter along with a sequence of iterables and returns an output after applying the function to each iterable present in the sequence.

Syntax:

map(function, iterables)

Example without Map

my_pets = ['alfred', 'tabitha', 'william', 'arla']

uppered_pets = []

for pet in my_pets:

pet_=pet.upper()

uppered_pets.append(pet_)

print(uppered_pets)

Example with Map

my_pets = ['alfred', 'tabitha', 'william', 'arla']

uppered_pets=list(map(str.upper,my_pets)) print(uppered_pets)

//map with multiple list as input

circle_areas = [3.56773, 5.57668, 4.00914, 56.24241, 9.01344, 32.00013]

result = list(map(round, circle_areas, range(1,7)))

print(result)

filter() Function

- filter extracts each element in the sequence for which the function returns True.
- filter(), first of all, requires the function to return boolean values (true or false) and then passes each element in the iterable through the function, "filtering" away those that are false

Syntax:

filter(func, iterable)

The following points are to be noted regarding filter():

- Unlike map(), only one iterable is required.
- The func argument is required to return a boolean type. If it doesn't, filter simply returns the iterable passed to it. Also, as only one iterable is required, it's implicit that func must only take one argument.
- filter passes each element in the iterable through func and returns only the ones that evaluate to true. I mean, it's right there in the name -- a "filter".

Example:

def isOdd(x): return x % 2 == 1

filter(isOdd, [1, 2, 3, 4])

# output ---> [1, 3]

filter() Function

Example:

Python 3

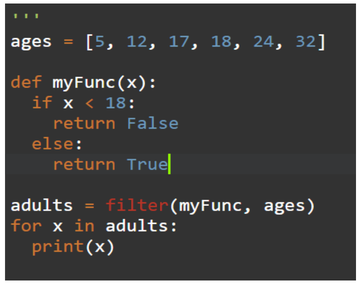
scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]

def is_A_student(score):

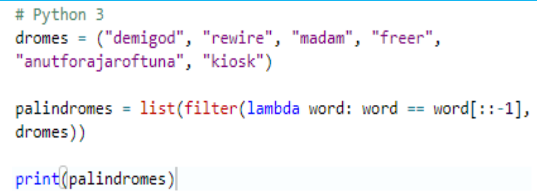
return score > 75

over_75 = list(filter(is_A_student, scores))

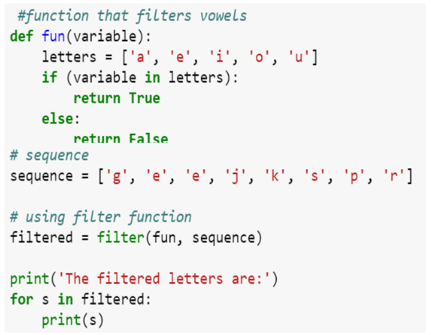
print(over_75)



```
'''
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)
for x in adults:
  print(x)
```



```
# Python 3
dromes = ("demigod", "rewire", "madam", "freer",
"anutforajaroftuna", "kiosk")

palindromes = list(filter(lambda word: word == word[::-1],
dromes))

print(palindromes)
```



```
#function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

reduce() Function

- reduce, combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an initial value that initializes the reduction, and that ends up being the return value if the list is empty.
- The “reduce” function will transform a given list into a single value by applying a given function continuously to all the elements. It basically keeps operating on pairs of elements until there are no more elements left.
- reduce applies a function of two arguments cumulatively to the elements of an iterable, optionally starting with an initial argument

Syntax:

```
reduce(func, iterable[, initial])
```

Example:

```
reduce(lambda s,x: s+str(x), [1, 2, 3, 4], '')
#output '1234‘
my_list = [3,8,4,9,5]
reduce(lambda a, b: a * b, my_list)
#output 4320 ( 3*8*4*9*5)
```

```
from functools import reduce
y = filter(lambda x: (x>=3), (1,2,3,4))
print(list(y))

reduce(lambda a,b: a+b,[23,21,45,98])

nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16,
    29, 21, 60, 27, 62, 59, 86, 56]
sum = reduce(lambda x, y : x + y, nums) / len(nums)
```

map(), filter() and reduce() Function

Using filter() within map():

```
c = map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4)))
print(list(c))
```

Using map() within filter():

```
c = filter(lambda x: (x>=3),map(lambda x:x+x, (1,2,3,4))) #lambda x: (x>=3)
print(list(c))
```

Using map() and filter() within reduce():

```
d = reduce(lambda x,y: x+y,map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4))))
print(d)
```

map(), filter() and reduce() Function

```
from functools import reduce
# Use map to print the square of each numbers rounded# to two decimal places
my_floats = [4.35, 6.09, 3.25, 9.77, 2.16, 8.88, 4.59]
# Use filter to print only the names that are less than or equal to seven letters
my_names = ["olumide", "akinremi", "josiah", "temidayo", "omoseun"]
# Use reduce to print the product of these numbers
my_numbers = [4, 6, 9, 23, 5]
map_result = list(map(lambda x: round(x ** 2, 3), my_floats))
filter_result = list(filter(lambda name: len(name) <= 7, my_names))
reduce_result = reduce(lambda num1, num2: num1 * num2, my_numbers)
print(map_result)
print(filter_result)
print(reduce_result)
```

Function vs Procedure

S.No	Functional Paradigms	Procedural Paradigm
1	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Derived from structured programming, based on the concept of modular programming or the *procedure call*
2	Main traits are Lambda alculus, compositionality, formula, recursion, referential transparency	Main traits are Local variables, sequence, selection, iteration, and modularization
3	**Functional** programming focuses on **expressions**	**Procedural** programming focuses on **statements**
4	Often recursive. Always returns the same output for a given input.	The output of a routine does not always have a direct correlation with the input.
5	Order of evaluation is usually undefined.	Everything is done in a specific order.
6	Must be stateless. i.e. No operation can have side effects.	Execution of a routine may have side effects.
7	Good fit for parallel execution, Tends to emphasize a divide and conquer approach.	Tends to emphasize implementing solutions in a linear fashion.

Function vs Object Oriented

S.No	Functional Paradigms	Object Oriented Paradigm
1	FP uses Immutable data.	OOP uses Mutable data.
2	Follows Declarative Programming based Model.	Follows Imperative Programming Model.
3	What it focuses is on: "What you are doing. in the programme."	What it focuses is on "How you are doing your programming."
4	Supports Parallel Programming.	No supports for Parallel Programming.
5	Its functions have no-side effects.	Method can produce many side effects.
6	Flow Control is performed using function calls & function calls with recursion.	Flow control process is conducted using loops and conditional statements.
7	Execution order of statements is not very important.	Execution order of statements is important.
8	Supports both "Abstraction over Data" and "Abstraction over Behavior."	Supports only "Abstraction over Data".

GUI & Event Handling Programming Paradigm

Event Driven Programming Paradigm

- Event-driven programming is a programming paradigm in which the flow of program execution is determined by events - for example a user action such as a mouse click, key press, or a message from the operating system or another program.
- An event-driven application is designed to detect events as they occur, and then deal with them using an appropriate event-handling procedure.
- In a typical modern event-driven program, there is no discernible flow of control. The main routine is an event-loop that waits for an event to occur, and then invokes the appropriate event-handling routine.
- Event callback is a function that is invoked when something significant happens like when click event is performed by user or the result of database query is available.

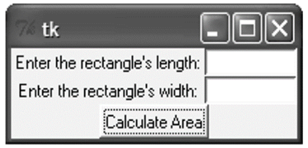
Event Handlers: Event handlers is a type of function or method that run a specific action when a specific event is triggered. For example, it could be a button that when user click it, it will display a message, and it will close the message when user click the button again, this is an event handler.

Trigger Functions: Trigger functions in event-driven programming are a functions that decide what code to run when there are a specific event occurs, which are used to select which event handler to use for the event when there is specific event occurred.

Events: Events include mouse, keyboard and user interface, which events need to be triggered in the program in order to happen, that mean user have to interacts with an object in the program, for example, click a button by a mouse, use keyboard to select a button and etc.

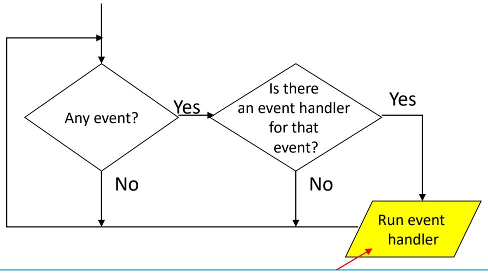
Introduction

- A graphical user interface allows the user to interact with the operating system and other programs using graphical elements such as icons, buttons, and dialog boxes.
- GUIs popularized the use of the mouse.
- GUIs allow the user to point at graphical elements and click the mouse button to activate them.
- GUI Programs Are Event-Driven
- User determines the order in which things happen
- GUI programs respond to the actions of the user, thus they are event driven.
- The tkinter module is a wrapper around tk, which is a wrapper around tcl, which is what is used to create windows and graphical user interfaces.



Introduction

- A major task that a GUI designer needs to do is to determine what will happen when a GUI is invoked
- Every GUI component may generate different kinds of “events” when a user makes access to it using his mouse or keyboard
- E.g. if a user moves his mouse on top of a button, an event of that button will be generated to the Windows system
- E.g. if the user further clicks, then another event of that button will be generated (actually it is the click event)
- For any event generated, the system will first check if there is an event handler, which defines the action for that event
- For a GUI designer, he needs to develop the event handler to determine the action that he wants Windows to take for that event.



GUI Using Python

- Tkinter: Tkinter is the Python interface to the Tk GUI toolkit shipped with Python.
- wxPython: This is an open-source Python interface for wxWindows
- PyQt –This is also a Python interface for a popular cross-platform Qt GUI library.
- JPython: JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine

Tkinter Programming

- Tkinter is the standard GUI library for Python.
- Creating a GUI application using Tkinter

Steps

- Import the Tkinter module.

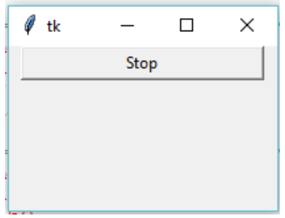
  Import tKinter as tk
- Create the GUI application main window.

  root = tk.Tk()
- Add one or more of the above-mentioned widgets to the GUI application.

  button = tk.Button(root, text='Stop', width=25, command=root.destroy)

  button.pack()
- Enter the main event loop to take action against each event triggered by the user.

  root.mainloop()

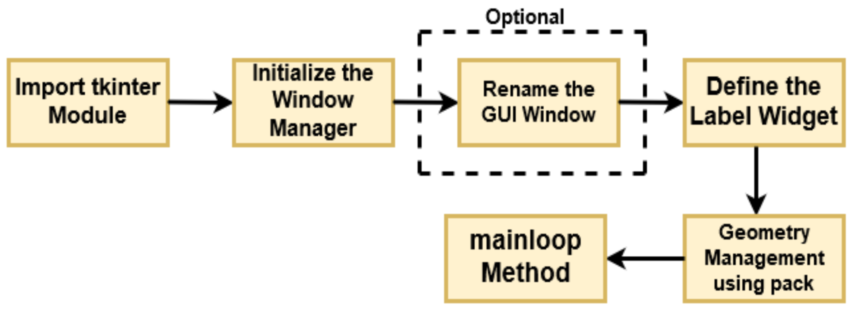


Tkinter widgets

- Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

Widget	**Description**
Label	Used to contain text or images
Button	Similar to a Label but provides additional functionality for mouse overs, presses, and releases as well as keyboard activity/events
Canvas	Provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps
Radiobutton	Set of buttons of which only one can be "pressed" (similar to HTML radio input)
Checkbutton	Set of boxes of which any number can be "checked" (similar to HTML checkbox input)
Entry	Single-line text field with which to collect keyboard input (similar to HTML text input)
Frame	Pure container for other widgets
Listbox	Presents user list of choices to pick from
Menu	Actual list of choices "hanging" from a Menubutton that the user can choose from
Menubutton	Provides infrastructure to contain menus (pulldown, cascading, etc.)
Message	Similar to a Label, but displays multi-line text
Scale	Linear "slider" widget providing an exact value at current setting; with defined starting and ending values
Text	Multi-line text field with which to collect (or display) text from user (similar to HTML TextArea)
Scrollbar	Provides scrolling functionality to supporting widgets, i.e., Text, Canvas, Listbox, and Entry
Toplevel	Similar to a Frame, but provides a separate window container

Operation Using Tkinter Widget



Geometry Managers

- The pack() Method – This geometry manager organizes widgets in blocks before placing them in the parent widget.

  widget.pack(pack_options)

  options

  - expand – When set to true, widget expands to fill any space not otherwise used in widget's parent.
  - fill – Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).
  - side – Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.
- The grid() Method – This geometry manager organizes widgets in a table-like structure in the parent widget.

  widget.grid(grid_options)

  options –

  - Column/row – The column or row to put widget in; default 0 (leftmost column).
  - Columnspan, rowsapn– How many columns or rows to widgetoccupies; default 1.
  - ipadx, ipady – How many pixels to pad widget, horizontally and vertically, inside widget's borders.
  - padx, pady – How many pixels to pad widget, horizontally and vertically, outside v's borders.

Geometry Managers

- The place() Method – This geometry manager organizes widgets by placing them in a specific position in the parent widget.

  widget.place(place_options)

  options –

  - anchor – The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW (the upper left corner of widget)
  - bordermode – INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.
  - height, width – Height and width in pixels.
  - relheight, relwidth – Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
  - relx, rely – Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
  - x, y – Horizontal and vertical offset in pixels.

Common Widget Properties

Common attributes such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors
- Relief styles
- Bitmaps
- Cursors

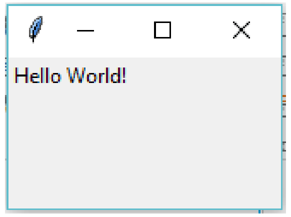
Label Widgets

- A label is a widget that displays text or images, typically that the user will just view but not otherwise interact with. Labels are used for such things as identifying controls or other parts of the user interface, providing textual feedback or results, etc.
- Syntax

  tk.Label(parent,text=“message”)

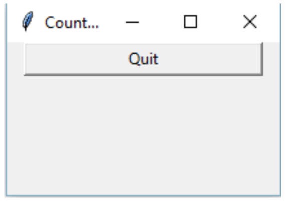
Example:

```
import tkinter as tk
root = tk.Tk()
label = tk.Label(root, text='Hello World!')
label.grid()
root.mainloop()
```



Button Widgets

```
import tkinter as tk
r = tk.Tk()
r.title('Counting Seconds')
button = tk.Button(r, text='Stop', width=25, command=r.destroy)
button.pack()
r.mainloop()
```



Button Widgets

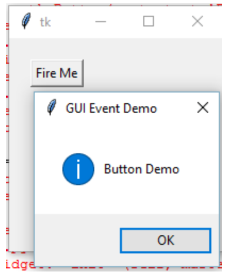
- A button, unlike a frame or label, is very much designed for the user to interact with, and in particular, press to perform some action. Like labels, they can display text or images, but also have a whole range of new options used to control their behavior.

Syntax

button = ttk.Button(parent, text=‘ClickMe', command=submitForm)

Example:

```
import tkinter as tk
from tkinter import messagebox
def hello():
  msg = messagebox.showinfo( "GUI Event Demo","Button Demo")
root = tk.Tk()
root.geometry("200x200")
b = tk.Button(root, text='Fire Me',command=hello)
b.place(x=50,y=50)
root.mainloop()
```



Button Widgets

- Button:To add a button in your application, this widget is used.

Syntax :

w=Button(master, text=“caption” option=value)

- master is the parameter used to represent the parent window.
- activebackground: to set the background color when button is under the cursor.
- activeforeground: to set the foreground color when button is under the cursor.
- bg: to set he normal background color.
- command: to call a function.
- font: to set the font on the button label.
- image: to set the image on the button.
- width: to set the width of the button.
- height: to set the height of the button.

Entry Widgets

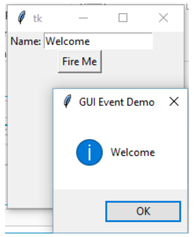
- An entry presents the user with a single line text field that they can use to type in a string value. These can be just about anything: their name, a city, a password, social security number, and so on.

Syntax

name = ttk.Entry(parent, textvariable=username)

Example:

```
def hello():
   msg = messagebox.showinfo( "GUI Event Demo",t.get())
root = tk.Tk()
root.geometry("200x200")
l1=tk.Label(root,text="Name:")
l1.grid(row=0)
t=tk.Entry(root)
t.grid(row=0,column=1)
b = tk.Button(root, text='Fire Me',command=hello)
b.grid(row=1,columnspan=2);
root.mainloop()
```



Canvas

- The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.
- It is used to draw pictures and other complex layout like graphics, text and widgets.

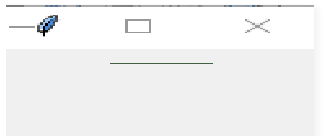
Syntax:

w = Canvas(master, option=value)

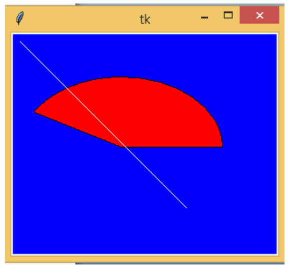
- master is the parameter used to represent the parent window.
- bd: to set the border width in pixels.
- bg: to set the normal background color.
- cursor: to set the cursor used in the canvas.
- highlightcolor: to set the color shown in the focus highlight.
- width: to set the width of the widget.
- height: to set the height of the widget.

Canvas

```
from tkinter import *
master = Tk()
w = Canvas(master, width=40, height=60)
w.pack()
canvas_height=20
canvas_width=200
y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y )
mainloop()
```



```
from tkinter import *
from tkinter import messagebox
top = Tk()
C = Canvas(top, bg = "blue", height = 250, width = 300)
coord = 10, 50, 240, 210
arc = C.create_arc(coord, start = 0, extent = 150, fill = "red")
line = C.create_line(10,10,200,200,fill = 'white')
C.pack()
top.mainloop()
```



Checkbutton

- A checkbutton is like a regular button, except that not only can the user press it, which will invoke a command callback, but it also holds a binary value of some kind (i.e. a toggle). Checkbuttons are used all the time when a user is asked to choose between, e.g. two different values for an option.

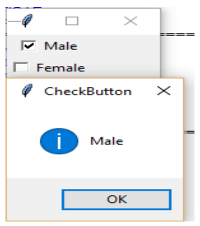
Syntax

w = CheckButton(master, option=value)

Example:

```
from tkinter import *
root= Tk()
root.title('Checkbutton Demo')
v1=IntVar()
v2=IntVar()
cb1=Checkbutton(root,text='Male', variable=v1,onvalue=1, offvalue=0, command=test)
cb1.grid(row=0)
cb2=Checkbutton(root,text='Female', variable=v2,onvalue=1, offvalue=0, command=test)
cb2.grid(row=1)
root.mainloop()
```

```
def test():
   if(v1.get()==1 ):
      v2.set(0)
      print("Male")
   if(v2.get()==1):
      v1.set(0)
      print("Female")
```



radiobutton

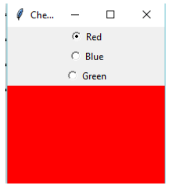
- A radiobutton lets you choose between one of a number of mutually exclusive choices; unlike a checkbutton, it is not limited to just two choices. Radiobuttons are always used together in a set and are a good option when the number of choices is fairly small
- **Syntax**

    w = CheckButton(master, option=value)

Example:

```
    root= Tk()
    root.geometry("200x200")
    radio=IntVar()
    rb1=Radiobutton(root,text='Red', variable=radio,width=25,value=1, command=choice)
    rb1.grid(row=0)
    rb2=Radiobutton(root,text='Blue', variable=radio,width=25,value=2, command=choice)
    rb2.grid(row=1)
    rb3=Radiobutton(root,text='Green', variable=radio,width=25,value=3, command=choice)
    rb3.grid(row=3)
    root.mainloop()
```

```
def choice():
   if(radio.get()==1):
      root.configure(background='red')
   elif(radio.get()==2):
      root.configure(background='blue')
   elif(radio.get()==3):
      root.configure(background='green')
```



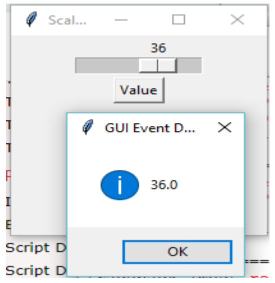
Scale

- Scale widget is used to implement the graphical slider to the python application so that the user can slide through the range of values shown on the slider and select the one among them. We can control the minimum and maximum values along with the resolution of the scale. It provides an alternative to the Entry widget when the user is forced to select only one value from the given range of values.
- **Syntax**

    w = Scale(top, options)

Example:

```
from tkinter import messagebox
root= Tk()
root.title('Scale Demo')
root.geometry("200x200")
def slide():
   msg = messagebox.showinfo( "GUI Event Demo",v.get())
v = DoubleVar()
scale = Scale( root, variable = v, from_ = 1, to = 50, orient = HORIZONTAL)
scale.pack(anchor=CENTER)
btn = Button(root, text="Value", command=slide)
btn.pack(anchor=CENTER)
root.mainloop()
```



Spinbox

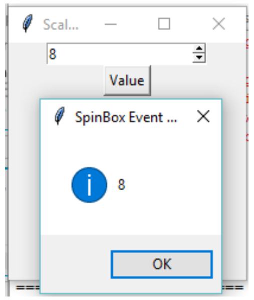
- The Spinbox widget is an alternative to the Entry widget. It provides the range of values to the user, out of which, the user can select the one.

Syntax

    w = Spinbox(top, options)

Example:

```
from tkinter import *
from tkinter import messagebox

root= Tk()
root.title('Scale Demo')
root.geometry("200x200")

def slide():
   msg = messagebox.showinfo( "SpinBox Event Demo",spin.get())

spin = Spinbox(root, from_= 0, to = 25)
spin.pack(anchor=CENTER)
btn = Button(root, text="Value", command=slide)
btn.pack(anchor=CENTER)
root.mainloop()
```



Menubutton

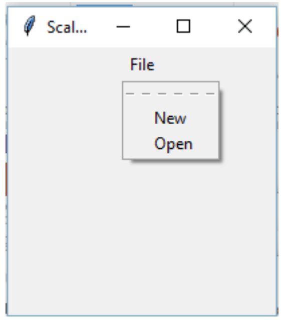
- Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

Syntax

    w = Menubutton(Top, options)

Example:

```
from tkinter import *
from tkinter import messagebox
root= Tk()
root.title('Scale Demo')
root.geometry("200x200")
menubutton = Menubutton(root, text = "File", relief = FLAT)
menubutton.grid()
menubutton.menu = Menu(menubutton)
menubutton["menu"]=menubutton.menu
menubutton.menu.add_checkbutton(label = "New", variable=IntVar(),command=)
menubutton.menu.add_checkbutton(label = "Open", variable = IntVar())
menubutton.pack()
root.mainloop()
```



Menubutton

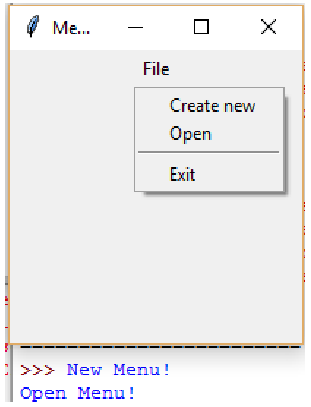
- Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

Syntax

w = Menubutton(Top, options)

Example:

```
from tkinter import *
from tkinter import messagebox
root= Tk()
root.title('Menu Demo')
root.geometry("200x200")
def new():
   print("New Menu!")
def disp():
   print("Open Menu!")
menubutton = Menubutton(root, text="File")
menubutton.grid()
menubutton.menu = Menu(menubutton, tearoff = 0)
menubutton["menu"] = menubutton.menu
menubutton.menu.add_command(label="Create new",command=new)
menubutton.menu.add_command(label="Open",command=disp)
menubutton.menu.add_separator()
menubutton.menu.add_command(label="Exit",command=root.quit)
menubutton.pack()
```

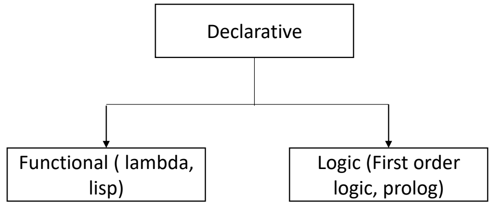


```
>>> New Menu!
Open Menu!
```

Declarative Programming Paradigm

Introduction

- Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow.
- This paradigm often considers programs as theories of a formal logic, and computations as deductions in that logic space.
- Declarative programming is often defined as any style of programming that is not imperative.
- Common declarative languages include those of database query languages (SQL), logic programming, functional programming, etc.
- A program that describes what computation should be performed and not how to compute it. Non-imperative, non-procedural.
- Any programming language that lacks side effects(example: a function might modify a global variable or static variable, modify one of its arguments, raise an exception,).
- A language with a clear correspondence to mathematical logic.



Declarative Programming Paradigm

- A program that describes what computation should be performed and not how to compute it .
- Any programming language that lacks side effects (or more specifically, is referentially transparent).
- A language with a clear correspondence to mathematical logic
- Here, the term side effect was mentioned.
- A function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world. For example, a function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, etc.

- Examples of declarative languages are HTML, XML, CSS, JSON and SQL, and there are more

SQL Elements

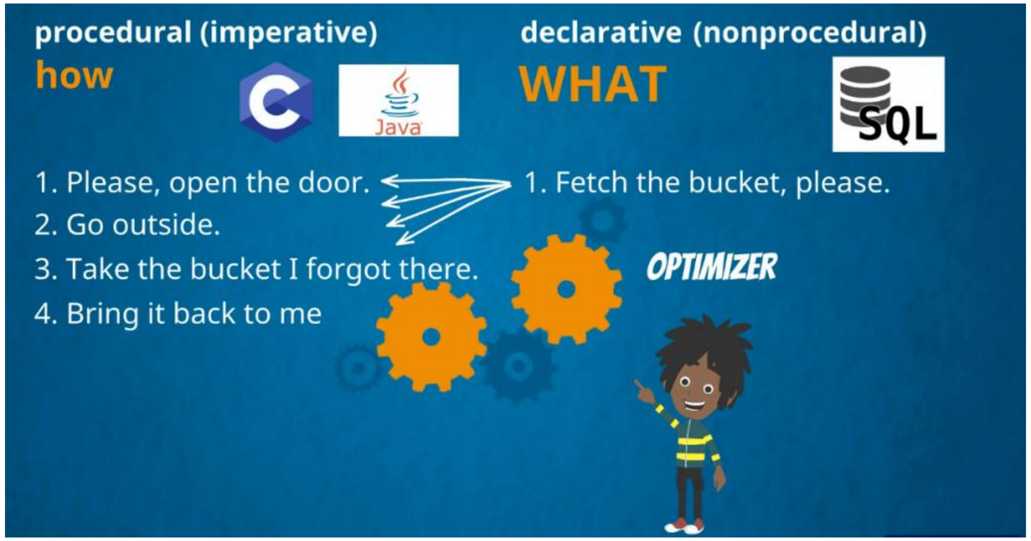
SQL is the standard language used to communicate with a relational database.

It can be used to retrieve data from a database using a query but it can also be used to create, manage, destroy as well as modify their structure and contents.

The language is subdivided into several language elements, including:

- Clauses
- Expressions
- Predicates
- Queries
- Statements

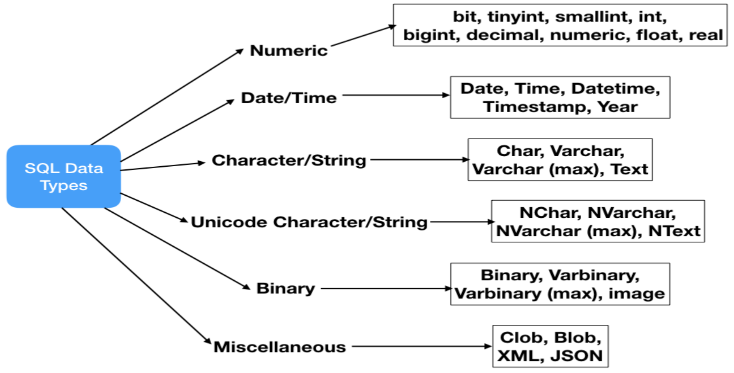
Procedure vs declarative



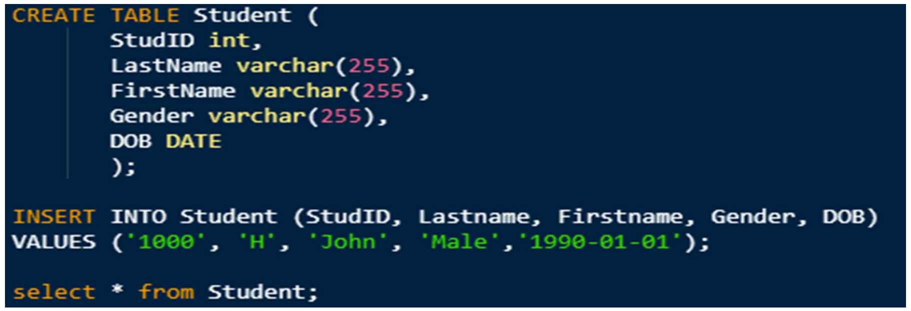
1. Please, open the door.
2. Go outside.
3. Take the bucket I forgot there.
4. Bring it back to me

1. Fetch the bucket, please.

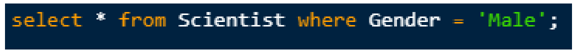
SQL Data Types


SQL Data Types
Numeric
bit, tinyint, smallint, int, bigint, decimal, numeric, float, real
Date/Time
Date, Time, Datetime, Timestamp, Year
Character/String
Char, Varchar, Varchar (max), Text
Unicode Character/String
NChar, NVarchar, NVarchar (max), NText
Binary
Binary, Varbinary, Varbinary (max), image
Miscellaneous
Clob, Blob, XML, JSON

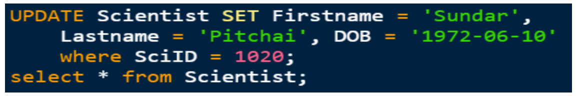
SQL Data Types



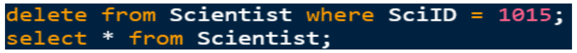
```
CREATE TABLE Student (
        StudID int,
        LastName varchar(255),
        FirstName varchar(255),
        Gender varchar(255),
        DOB DATE
        );

INSERT INTO Student (StudID, Lastname, Firstname, Gender, DOB)
VALUES ('1000', 'H', 'John', 'Male','1990-01-01');

select * from Student;
```



```
select * from Scientist where Gender = 'Male';
```



```
UPDATE Scientist SET Firstname = 'Sundar',
    Lastname = 'Pitchai', DOB = '1972-06-10'
    where SciID = 1020;
select * from Scientist;
```



```
delete from Scientist where SciID = 1015;
select * from Scientist;
```

SQL Data Types

```
def insert(rno,name):
   cur.execute("INSERT INTO stud VALUES(?,?,)", (no,name))
   conn.commit()
   print("Record Inserted")

def update(rno,name):
   cur.execute("update stud set name='"+name+"' where
rno='"+rno+"'")
   conn.commit()
   print('Record updated')

def select(rno):
   cur.execute("select * from stud")
   for row in cur.fetchAll():
          if(row[0]==rno):
                 print("Name ="+name)
def delete(rno):
   cur.execute("delete from stud where rno='"+rno+"'")
   conn.commit()
   print('Record deleted')
```

```
conn=sqlite3.connect("univ.db")
cur=conn.cursor()
sql ="create table stud23 ( regno varchar(10),name
varchar(20));"
cur.execute(sql)
insert('123,','bob')
insert('124,','danny')
update('124','daniel')
delete(124)
select(124
```

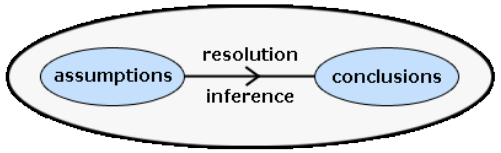
Logical Programming Paradigm

Logical Programming Paradigm

- It can be an abstract model of computation.
- Solve logical problems like puzzles, series
- Have knowledge base which we know before and along with the question you specify knowledge and how that knowledge is to be applied through a series of rules
- The Logical Paradigm takes a declarative approach to problem-solving.
- Various logical assertions about a situation are made, establishing all known facts.
- Then queries are made.

Logical Programming Paradigm

Logic programming is a paradigm where computation arises from proof search in a logic according to a fixed, predictable strategy. A logic is a language. It has syntax and semantics. It. More than a language, it has inference rules.



Syntax:

Syntax: the rules about how to form formulas; usually the easy part of a logic.

Semantics Semantics:

About the meaning carried by the formulas, mainly in terms of logical consequences.

Inference rules:

Inference rules describe correct ways to derive describe correct ways to derive conclusions.

Logical Programming Paradigm

Logic :

A Logic program is a set of predicates. Ex parent, siblings

Predicates :

Define relations between their arguments. Logically, a Logic program states what holds. Each predicate has a name, and zero or more arguments. The predicate name is a atom. Each argument is an arbitrary Logic term. A predicate is defined by a collection of clauses.

Example: Mother(x,y)

Clause :

A clause is either a rule or a fact. The clauses that constitute a predicate denote logical alternatives: If any clause is true, then the whole predicate is true.

Example:

Mother(X,Y) <= female(X) & parent(X,Y) # implies X is the mother of Y, if y has to female

Parts of Logical Programming Paradigm

- A series of definitions/declarations that define the problem domain (fact)
- Statements of relevant facts (rules)
- Statement of goals in the form of a query (query)

Example:

Given information about fatherhood and motherhood, determine grand parent relationship

E.g. Given the information called facts

John is father of Lily

Kathy is mother of Lily

Lily is mother of Bill

Ken is father of Karen

Who are grand parents of Bill?

Who are grand parents of Karen?

Logical Programming Paradigm

Fact

A fact must start with a predicate (which is an atom). The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists.

Facts are axioms; relations between terms that are assumed to be true.

Example facts:

```
+big('bear')
+big('elephant')
+small('cat')
+brown('bear')
+black('cat')
+grey('elephant')
```

Consider the 3 fact saying 'cat' is a smallest animal and fact 6 saying the elephant is grey in color

Rule

Rules are theorems that allow new inferences to be made.

```
dark(X)<=black(X)
dark(X)<=brown(X)
```

Consider rule 1 saying the color is black its consider to be dark color.

Logical Programming Paradigm

Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be the same as that appears in the consulted program.

```
print(pyDatalog.ask('father_of(X,jess)'))
```

Output:

```
{('jack',)}
X
print(father_of(X,'jess'))
```

Output:

```
jack
X
```

Logical Programming Paradigm

```
from pyDatalog import pyDatalog
pyDatalog.create_atoms('parent,male,female,son,daughter,X,Y,Z')
+male('adam')
+female('anne')
+female('barney')
+male('james')
+parent('barney','adam')
+parent('james','anne')

#The first rule is read as follows: for all X and Y, X is the son of Y if there exists X and Y such that Y is the parent of X and X is male.
#The second rule is read as follows: for all X and Y, X is the daughter of Y if there exists X and Y such that Y is the parent of X and X is female.
son(X,Y)<= male(X) & parent(Y,X)
daughter(X,Y)<= parent(Y,X) & female(X)
print(pyDatalog.ask('son(adam,Y)'))
print(pyDatalog.ask('daughter(anne,Y)'))
print(son('adam',X))
```

Logical Programming Paradigm

```
pyDatalog.create_terms('factorial, N')
factorial[N] = N*factorial[N-1]
factorial[1] = 1
print(factorial[3]==N)
```

Logical Programming Paradigm

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, works_in, department_size, manager, indirect_manager, count_of_indirect_reports')
# Mary works in Production
+works_in('Mary', 'Production')
+works_in('Sam',  'Marketing')
+works_in('John', 'Production')
+works_in('John', 'Marketing')
+(manager['Mary'] == 'John')
+(manager['Sam']  == 'Mary')
+(manager['Tom']  == 'Mary')

indirect_manager(X,Y) <= (manager[X] == Y)
print(works_in(X,  'Marketing'))
indirect_manager(X,Y) <= (manager[X] == Z) & indirect_manager(Z,Y)
print(indirect_manager('Sam',X))
```

Logical Programming Paradigm

Lucy is a Professor

Danny is a Professor

James is a Lecturer

All professors are Dean

Write a Query to reterive all deans?

Soln

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,professor,lecturer, dean')
+professor('lucy')
+professor('danny')
+lecturer('james')
dean(X)<=professor(X)
print(dean(X))
```

Logical Programming Paradigm

```
likes(john, susie).                  /* John likes Susie */
likes(X, susie).                     /* Everyone likes Susie */
likes(john, Y).                      /* John likes everybody */
likes(john, Y), likes(Y, john).      /* John likes everybody and everybody likes John */
likes(john, susie); likes(john,mary). /* John likes Susie or John likes Mary */
not(likes(john,pizza)).              /* John does not like pizza */
likes(john,susie) :- likes(john,mary)./* John likes Susie if John likes Mary.

rules
friends(X,Y) :- likes(X,Y),likes(Y,X).          /* X and Y are friends if they like each other */
hates(X,Y) :- not(likes(X,Y)).                  /* X hates Y if X does not like Y. */
enemies(X,Y) :- not(likes(X,Y)),not(likes(Y,X)).  /* X and Y are enemies if they don't like each other */
```

Imperative Programming Paradigm

Imperative Programming Paradigm

- It’s a programming paradigm that describes computation as statements that change a program state.
- Imperative programs are a sequence of commands for the computer to perform.
- imperative programming paradigm assumes that the computer can maintain through environments of variables any changes in a computation process.
- Computations are performed through a guided sequence of steps, in which these variables are referred to or changed. The order of the steps is crucial, because a given step will have different consequences depending on the current values of variables when the step is executed.
- Popular programming languages are imperative more often than they are any other paradigm studies because the imperative paradigm most closely resembles the actual machine itself, so the programmer is much closer to the machine;
- Imperative programs define sequences of commands/statements for the computer that change a program state (i.e., set of variables)
  - Commands are stored in memory and executed in the order found
  - Commands retrieve data, perform a computation, and assign the result to a memory location

Imperative Programming Paradigm

- In a computer program, a variable stores the data. The contents of these locations at any given point in the program’s execution are called the program’s state. Imperative programming is characterized by programming with state and commands which modify the state.
- The first imperative programming languages were machine languages.
- Machine Language : Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory.
  - MOV AL, BL
  - MOV A, B

Imperative vs Declarative Programming Paradigm

- Declarative programming is a programming paradigm … that expresses the logic of a computation without describing its control flow. Its focus how to do a task
- Imperative programming is a programming paradigm that uses statements that change a program's state. It focus on what to day rather than how to do it.

Example:

#declarative

```
    small_nums = [ x for x in range(20) if x<5]
```

#Imperative

```
    small = []
    for I in range(20):
        if i<5:
            small.append(i)
```

Imperative Programming Paradigm

- Central elements of imperative paradigm:
  - Assignment statement: assigns values to memory locations and changes the current state of a program
  - Variables refer to memory locations
  - Step-by-step execution of commands
  - Control-flow statements: Conditional and unconditional (GO TO) branches and loops to change the flow of a program
- Example of computing the factorial of a number:

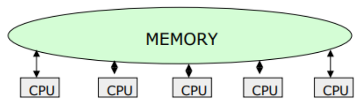
```
    unsigned int n = 5;
    unsigned int result = 1;
    while(n > 1)
    {
        result *= n;
        n--;
    }
```

Parallel & Concurrent Programming Paradigm

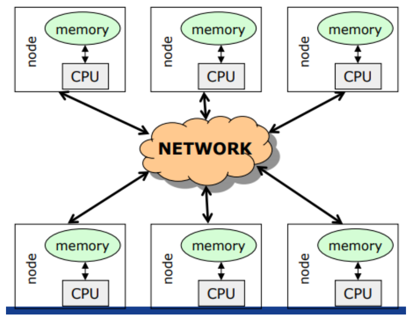
Introduction

- A system is said to be parallel if it can support two or more actions executing simultaneously i.e., multiple actions are simultaneously executed in parallel systems.
- The evolution of parallel processing, even if slow, gave rise to a considerable variety of programming paradigms.
- Parallelism Types:
  - Explicit Parallelism
  - Implicit Parallelism

Shared memory Architecture



Message Passing Architecture



Explicit parallelism

- Explicit Parallelism is characterized by the presence of explicit constructs in the programming language, aimed at describing (to a certain degree of detail) the way in which the parallel computation will take place.
- A wide range of solutions exists within this framework. One extreme is represented by the ``ancient'' use of basic, low level mechanisms to deal with parallelism--like fork/join primitives, semaphores, etc--eventually added to existing programming languages. Although this allows the highest degree of flexibility (any form of parallel control can be implemented in terms of the basic low level primitives gif), it leaves the additional layer of complexity completely on the shoulders of the programmer, making his task extremely complicate.

Implicit Parallelism

- Allows programmers to write their programs without any concern about the exploitation of parallelism. Exploitation of parallelism is instead automatically performed by the compiler and/or the runtime system. In this way the parallelism is transparent to the programmer maintaining the complexity of software development at the same level of standard sequential programming.
- Extracting parallelism implicitly is not an easy task. For imperative programming languages, the complexity of the problem is almost prohibitively and allows positive results only for restricted sets of applications (e.g., applications which perform intensive operations on arrays.
- Declarative Programming languages, and in particular Functional and Logic languages, are characterized by a very high level of abstraction, allowing the programmer to focus on what the problem is and leaving implicit many details of how the problem should be solved.
- Declarative languages have opened new doors to automatic exploitation of parallelism. Their focusing on a high level description of the problem and their mathematical nature turned into positive properties for implicit exploitation of parallelism.

Methods for parallelism

There are many methods of programming parallel computers. Two of the most common are message passing and data parallel.

1. Message Passing - the user makes calls to libraries to explicitly share information between processors.
2. Data Parallel - data partitioning determines parallelism
3. Shared Memory - multiple processes sharing common memory space
4. Remote Memory Operation - set of processes in which a process can access the memory of another process without its participation
5. Threads - a single process having multiple (concurrent) execution paths
6. Combined Models - composed of two or more of the above.

Methods for parallelism

Message Passing:

- Each Processor has direct access only to its local memory
- Processors are connected via high-speed interconnect
- Data exchange is done via explicit processor-to-processor communication i.e processes communicate by sending and receiving messages : send/receive messages
- Data transfer requires cooperative operations to be performed by each process (a send operation must have matching receive)

Data Parallel:

- Each process works on a different part of the same data structure
- Processors have direct access to global memory and I/O through bus or fast switching network
- Each processor also has its own memory (cache)
- Data structures are shared in global address space
- Concurrent access to shared memory must be coordinate
- All message passing is done invisibly to the programmer

Steps in Parallelism

- Independently from the specific paradigm considered, in order to execute a program which exploits parallelism, the programming language must supply the means to:
  - Identify parallelism, by recognizing the components of the program execution that will be (potentially) performed by different processors;
  - Start and stop parallel executions;
  - Coordinate the parallel executions (e.g., specify and implement interactions between concurrent components).

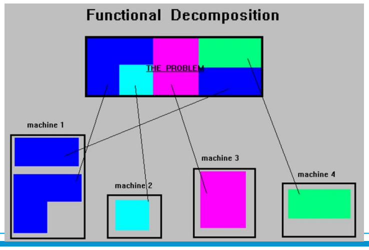
Ways for Parallelism

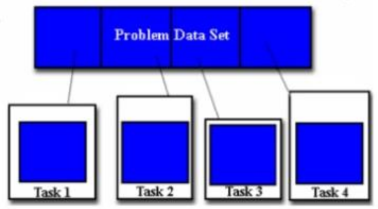
Functional Decomposition (Functional Parallelism)

- Decomposing the problem into different tasks which can be distributed to multiple processors for simultaneous execution
- Good to use when there is not static structure or fixed determination of number of calculations to be performed

Domain Decomposition (Data Parallelism)

- Partitioning the problem's data domain and distributing portions to multiple processors for simultaneous execution
- Good to use for problems where:
- data is static (factoring and solving large matrix or finite difference calculations)
- dynamic data structure tied to single entity where entity can be subsetted (large multi-body problems)
- domain is fixed but computation within various regions of the domain is dynamic (fluid vortices models)





Parallel Programming Paradigm

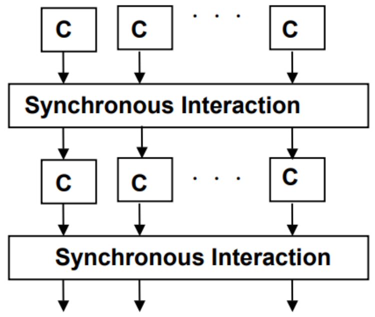
- Phase parallel
- Divide and conquer
- Pipeline
- Process farm
- Work pool

Note:

- The parallel program consists of number of super steps, and each super step has two phases : computation phase and interaction phase

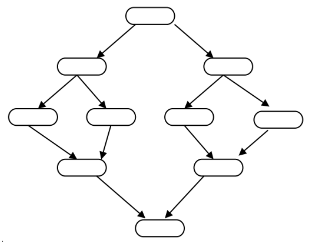
Phase Parallel Model

- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of supersteps, and each has two phases.
  - In a computation phase, multiple processes each perform an independent computation C.
  - In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.
  - Then next superstep is executed.



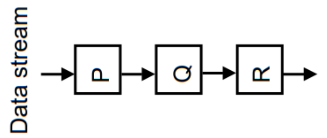
Divide and Conquer & Pipeline model

- A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
- The child processes then compute their workload in parallel and the results are merged by the parent.
- The dividing and the merging procedures are done recursively.
- This paradigm is very natural for computations such as quick sort.



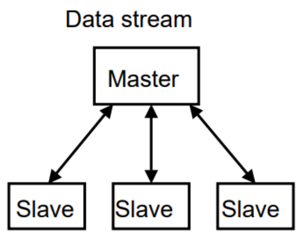
Pipeline

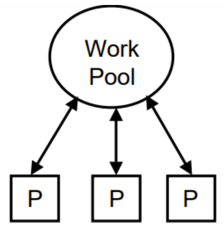
- In pipeline paradigm, a number of processes form a virtual pipeline.
- A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.



Process Farm & Work Pool Model

- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.
- This paradigm is often used in a shared variable model.
- A pool of works is realized in a global data structure.
- A number of processes are created. Initially, there may be just one piece of work in the pool.
- Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces put into the pool. The parallel program ends when the work pool becomes empty.
- This paradigm facilitates load balancing, as the workload is dynamically allocated to free processes.





Parallel Program using Python

- A thread is basically an independent flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. For Example, when you are playing a game say FIFA on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc.
- Threading is that it allows a user to run different parts of the program in a concurrent manner and make the design of the program simpler.
- Multithreading in Python can be achieved by importing the threading module.

Example:

```
import threading
from threading import *
```

Parallel program using Threads in Python

```
# simplest way to use a Thread is to instantiate it with a target
function and call start() to let it begin working.
from threading import Thread,current_thread
print(current_thread().getName())
def mt():
    print("Child Thread")
    for i in range(11,20):
        print(i*2)
def disp():
    for i in range(10):
        print(i*2)
child=Thread(target=mt)
child.start()
disp()
print("Executing thread name :",current_thread().getName())
```

```
from threading import Thread,current_thread
class mythread(Thread):
    def run(self):
        for x in range(7):
            print("Hi from child")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
```

Parallel program using Process in Python

```
import multiprocessing
def worker(num):
    print('Worker:', num)
    for i in range(num):
        print(i)
    return

jobs = []
for i in range(1,5):
    p = multiprocessing.Process(target=worker, args=(i+10,))
    jobs.append(p)
    p.start()
```

Concurrent Programming Paradigm

- Computing systems model the world, and the world contains actors that execute independently of, but communicate with, each other. In modelling the world, many (possibly) parallel executions have to be composed and coordinated, and that's where the study of concurrency comes in.
- There are two common models for concurrent programming: shared memory and message passing.
  - **Shared memory.** In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.
  - **Message passing.** In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling

Issues Concurrent Programming Paradigm

Concurrent programming is programming with multiple tasks. The major issues of concurrent programming are:

- Sharing computational resources between the tasks;
- Interaction of the tasks.

Objects shared by multiple tasks have to be safe for concurrent access. Such objects are called protected. Tasks accessing such an object interact with each other indirectly through the object.

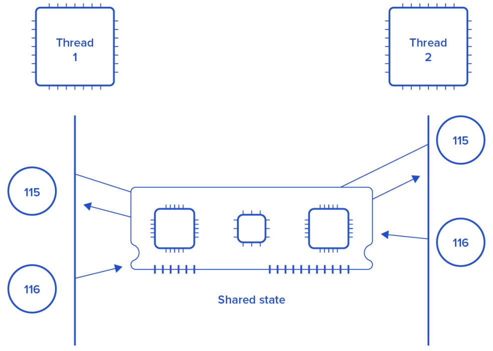
An access to the protected object can be:

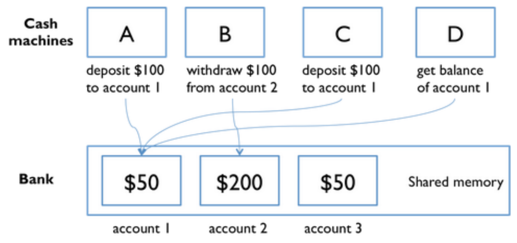
- Lock-free, when the task accessing the object is not blocked for a considerable time;
- Blocking, otherwise.

Blocking objects can be used for task synchronization. To the examples of such objects belong:

- Events;
- Mutexes and semaphores;
- Waitable timers;
- Queues

Issues Concurrent Programming Paradigm





Race Condition

```
import threading
x = 0    # A shared value
COUNT = 100

def incr():
   global x
   for i in range(COUNT):
      x += 1
      print(x)

def decr():
   global x
   for i in range(COUNT):
      x -= 1
      print(x)

t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
```

Synchronization in Python

Locks:

Locks are perhaps the simplest synchronization primitives in Python. A Lock has only two states — locked and unlocked (surprise). It is created in the unlocked state and has two principal methods — acquire() and release(). The acquire() method locks the Lock and blocks execution until the release() method in some other co-routine sets it to unlocked.

R-Locks:

R-Lock class is a version of simple locking that only blocks if the lock is held by another thread. While simple locks will block if the same thread attempts to acquire the same lock twice, a re-entrant lock only blocks if another thread currently holds the lock.

Semaphore:

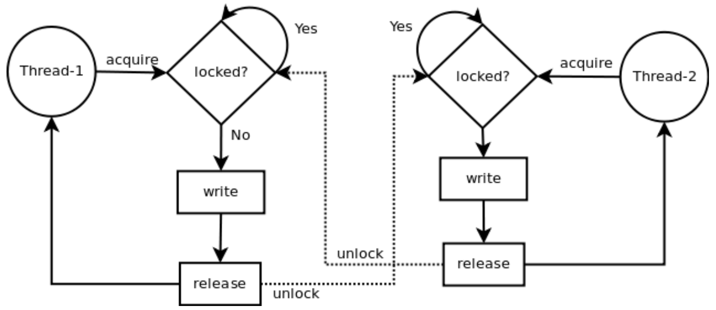
A semaphore has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore. Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously.

LOCK in python

Synchronization using LOCK

Locks have 2 states: locked and unlocked. 2 methods are used to manipulate them: acquire() and release(). Those are the rules:

1. if the state is unlocked: a call to acquire() changes the state to locked.
2. if the state is locked: a call to acquire() blocks until another thread calls release().
3. if the state is unlocked: a call to release() raises a RuntimeError exception.
4. if the state is locked: a call to release() changes the state to unlocked().



Synchronization in Python using Lock

```
import threading
x = 0    # A shared value
COUNT = 100
lock = threading.Lock()

def incr():
   global x
   lock.acquire()
   print("thread locked for increment cur x=",x)
   for i in range(COUNT):
      x += 1
      print(x)
   lock.release()
   print("thread release from increment cur x=",x)

def decr():
   global x
   lock.acquire()
   print("thread locked for decrement cur x=",x)
   for i in range(COUNT):
      x -= 1
      print(x)
   lock.release()
   print("thread release from decrement cur x=",x)
t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
```

Synchronization in Python using RLock

```
import threading

class Foo(object):
    lock = threading.RLock()
    def __init__(self):
        self.x = 0
    def add(self,n):
        with Foo.lock:
            self.x += n
    def incr(self):
        with Foo.lock:
            self.add(1)
    def decr(self):
        with Foo.lock:
            self.add(-1)


def adder(f,count):
    while count > 0:
        f.incr()
        count -= 1
```

```
def subber(f,count):
    while count > 0:
        f.decr()
        count -= 1

# Create some threads and make sure it works
COUNT = 10
f = Foo()
t1 = threading.Thread(target=adder,args=(f,COUNT))
t2 = threading.Thread(target=subber,args=(f,COUNT))
t1.start()
t2.start()
t1.join()
t2.join()
print(f.x)
```

Synchronization in Python using Semaphore

```
import threading
import time

done = threading.Semaphore(0)
item = None

def producer():
    global item
    print "I'm the producer and I produce data."
    print "Producer is going to sleep."
    time.sleep(10)
    item = "Hello"
    print "Producer is alive. Signaling the consumer."
    done.release()
```

```
def consumer():
    print "I'm a consumer and I wait for data."
    print "Consumer is waiting."
    done.acquire()
    print "Consumer got", item

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)
t1.start()
t2.start()
```

Synchronization in Python using event

```
import threading
import time
item = None
# A semaphore to indicate that an item is available
available = threading.Semaphore(0)
# An event to indicate that processing is complete
completed = threading.Event()
# A worker thread
def worker():
    while True:
        available.acquire()
        print "worker: processing", item
        time.sleep(5)
        print "worker: done"
        completed.set()
```

```
# A producer thread
def producer():
    global item
    for x in range(5):
        completed.clear()    # Clear the event
        item = x             # Set the item
        print "producer: produced an item"
        available.release()   # Signal on the semaphore
        completed.wait()
        print "producer: item was processed"
t1 = threading.Thread(target=producer)
t1.start()
t2 = threading.Thread(target=worker)
t2.setDaemon(True)
t2.start()
```

Producer and Consumer problem using thread

```
import threading,time,Queue
items = Queue.Queue()
# A producer thread
def producer():
    print "I'm the producer"
    for i in range(30):
        items.put(i)
        time.sleep(1)
# A consumer thread
def consumer():
    print "I'm a consumer", threading.currentThread().name
    while True:
        x = items.get()
        print threading.currentThread().name,"got", x
        time.sleep(5)
```

```
# Launch a bunch of consumers
cons = [threading.Thread(target=consumer)
        for i in range(10)]
for c in cons:
    c.setDaemon(True)
    c.start()
# Run the producer
producer()
```

Producer and Consumer problem using thread

```
import threading
import time
# A list of items that are being produced.  Note: it is actually
# more efficient to use a collections.deque() object for this.
items = []
# A condition variable for items
items_cv = threading.Condition()
def producer():
    print "I'm the producer"
    for i in range(30):
        with items_cv:        # Always must acquire the lock first
            items.append(i)    # Add an item to the list
            items_cv.notify()  # Send a notification signal
        time.sleep(1)
def consumer():
    print "I'm a consumer", threading.currentThread().name
    while True:
        with items_cv:         # Must always acquire the lock
            while not items:     # Check if there are any items
                items_cv.wait()  # If not, we have to sleep
            x = items.pop(0)     # Pop an item off
        print threading.currentThread().name,"got", x
        time.sleep(5)
cons = [threading.Thread(target=consumer)
        for i in range(10)]
for c in cons:
    c.setDaemon(True)
    c.start()
producer()
```

Network Programming Paradigm

Introduction

The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out. Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.

In the context of the Internet, clients are typically run on desktop or laptop computers attached to the Internet looking for information, whereas servers are typically run on larger computers with certain types of information available for the clients to retrieve. The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.

Network programming uses a particular type of network communication known as sockets. A socket is a software abstraction for an input or output medium of communication.

What is Socket?

- A socket is a software abstraction for an input or output medium of communication.
- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.
- A socket is the most vital and fundamental entity. Sockets are the end-point of a two-way communication link.
- An endpoint is a combination of IP address and the port number.

For Client-Server communication,

- Sockets are to be configured at the two ends to initiate a connection,
- Listen for incoming messages
- Send the responses at both ends
- Establishing a bidirectional communication.

Socket Types

Datagram Socket

- A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed. A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner. Unreliable means that information sent via datagrams isn't guaranteed to make it to its destination.

Stream Socket:

- A stream socket, or connected socket, is a socket through which data can be transmitted continuously. A stream socket is more akin to a live network, in which the communication link is continuously active. A stream socket is a "connected" socket through which data is transferred continuously.

Socket in Python

```
sock_obj = socket.socket( socket_family, socket_type, protocol=0)
```

socket_family: - Defines family of protocols used as transport mechanism.

Either AF_UNIX, or

AF_INET (IP version 4 or IPv4).

socket_type: Defines the types of communication between the two end-points.

SOCK_STREAM (for connection-oriented protocols, e.g., TCP), or

SOCK_DGRAM (for connectionless protocols e.g. UDP).

protocol: We typically leave this field or set this field to zero.

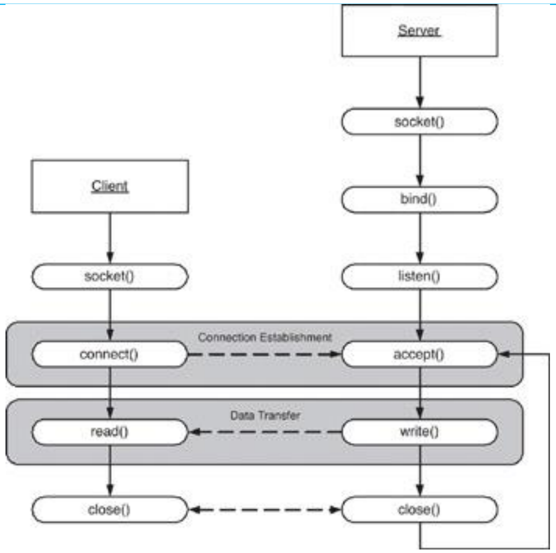
Example:

```
#Socket client example in python
import socket
#create an AF_INET, STREAM socket (TCP)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket Created'
```

Socket Creation

```
import socket
import sys
try:
        #create an AF_INET, STREAM socket (TCP)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, msg:
        print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
        sys.exit();

print 'Socket Created'
```

Client/server symmetry in Sockets applications



Socket in Python

To create a socket, we must use socket.socket() function available in the Python socket module, which has the general syntax as follows:

S = socket.socket(socket_family, socket_type, protocol=0)

socket_family: This is either AF_UNIX or AF_INET. We are only going to talk about INET sockets in this tutorial, as they account for at least 99% of the sockets in use.

socket_type: This is either SOCK_STREAM or SOCK_DGRAM.

Protocol: This is usually left out, defaulting to 0.

Client Socket Methods

Following are some client socket methods:

connect( ) : To connect to a remote socket at an address. An address format(host, port) pair is used for AF_INET address family.

Socket in Python

Server Socket Methods

bind( ): This method binds the socket to an address. The format of address depends on socket family mentioned above(AF_INET).

listen(backlog) : This method listens for the connection made to the socket. The backlog is the maximum number of queued connections that must be listened before rejecting the connection.

accept( ) : This method is used to accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair(conn, address) where conn is a new socket object which can be used to send and receive data on that connection, and address is the address bound to the socket on the other end of the connection.

General Socket in Python

sock_object.recv():

Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.

sock_object.send():

Apply this method to send messages from endpoints in case the protocol is TCP.

sock_object.recvfrom():

Call this method to receive messages at endpoints if the protocol used is UDP.

sock_object.sendto():

Invoke this method to send messages from endpoints if the protocol parameter is UDP.

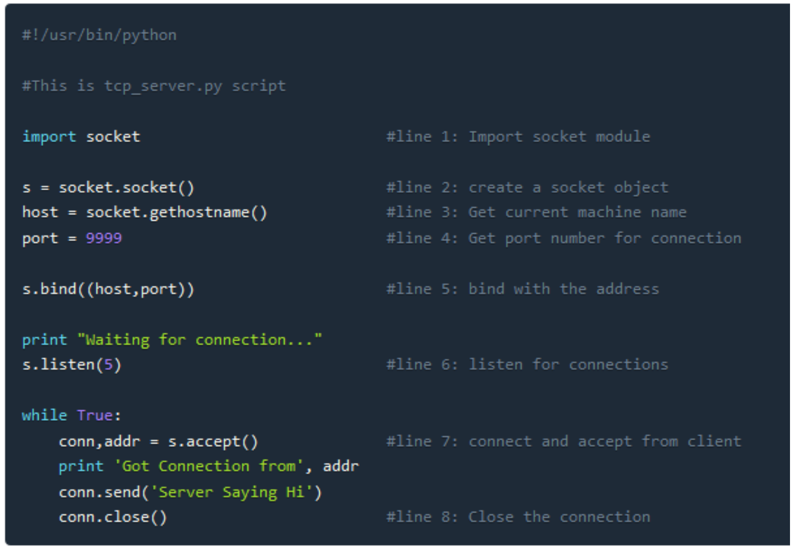
sock_object.gethostname():

This method returns hostname.

sock_object.close():

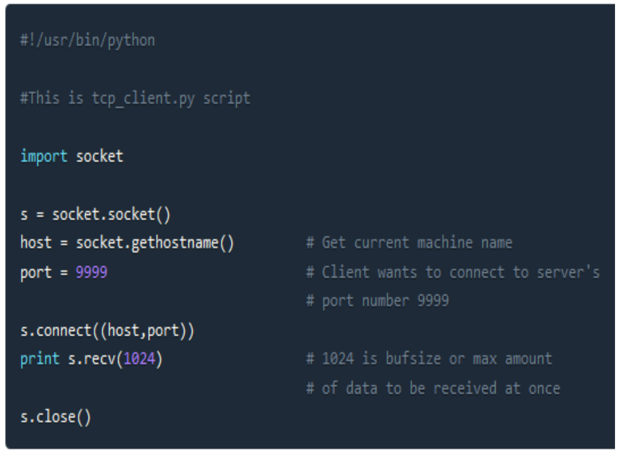
This method is used to close the socket. The remote endpoint will not receive data from this side.

Simple TCP Server



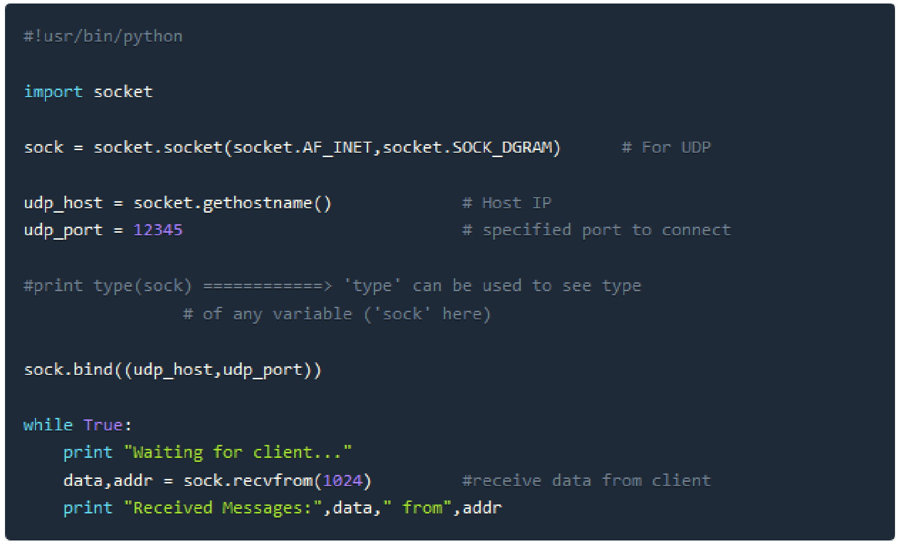
```
#!/usr/bin/python

#This is tcp_server.py script

import socket                           #line 1: Import socket module

s = socket.socket()                     #line 2: create a socket object
host = socket.gethostname()             #line 3: Get current machine name
port = 9999                             #line 4: Get port number for connection

s.bind((host,port))                     #line 5: bind with the address

print "Waiting for connection..."
s.listen(5)                             #line 6: listen for connections

while True:
    conn,addr = s.accept()              #line 7: connect and accept from client
    print 'Got Connection from', addr
    conn.send('Server Saying Hi')
    conn.close()                        #line 8: Close the connection
```

Simple TCP Client



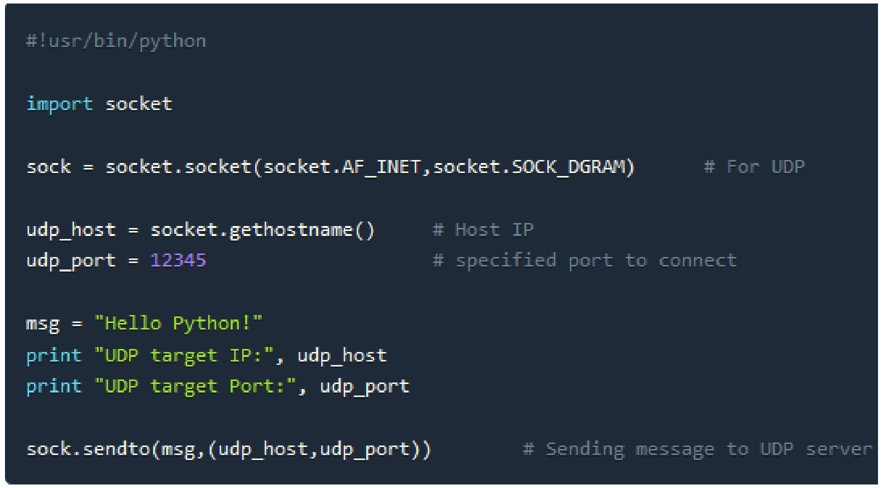
```
#!/usr/bin/python

#This is tcp_client.py script

import socket

s = socket.socket()
host = socket.gethostname()          # Get current machine name
port = 9999                          # Client wants to connect to server's
                                     # port number 9999

s.connect((host,port))
print s.recv(1024)                   # 1024 is bufsize or max amount
                                     # of data to be received at once

s.close()
```

Simple UDP Server



```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()             # Host IP
udp_port = 12345                            # specified port to connect

#print type(sock) ============> 'type' can be used to see type
                # of any variable ('sock' here)

sock.bind((udp_host,udp_port))

while True:
    print "Waiting for client..."
    data,addr = sock.recvfrom(1024)         #receive data from client
    print "Received Messages:",data," from",addr
```

Simple UDP Client



```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()     # Host IP
udp_port = 12345                    # specified port to connect

msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port

sock.sendto(msg,(udp_host,udp_port))        # Sending message to UDP server
```

Symbolic Programming Paradigm

Introduction

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

It Covers the following:

- As A calculator and symbols
- Algebraic Manipulations - Expand and Simplify
- Calculus – Limits, Differentiation, Series , Integration
- Equation Solving – Matrices

Calculator and Symbols

Rational - ½, or 5/2

```
>>import sympy as sym
>>a = sym.Rational(1, 2)
>>a                                      Answer will be 1/2
```

Constants like pi,e

```
>>sym.pi**2                              Answer is pi**2
>>sym.pi.evalf()                         Answer is 3.14159265358979
>> (sym.pi + sym.exp(1)).evalf()         Answer is 5.85987448204884
```

X AND Y

```
>> x = sym.Symbol('x')
>>y = sym.Symbol('y')
>>x + y + x – y                          Answer is 2*x
```

Algebraic Manipulations

EXPAND (X+Y)3 = X+3X^2Y+3XY^2+Y**

```
>> sym.expand((x + y) ** 3)                          Answer is x**3 + 3*x**2*y + 3*x*y**2 + y**3
>> 3 * x * y ** 2 + 3 * y * x ** 2 + x ** 3 + y ** 3  Answer is x**3 + 3*x**2*y + 3*x*y**2 + y**3
```

WITH TRIGNOMETRY LIKE SIN,COSINE

eg. COS(X+Y)= - SIN(X)*SIN(Y)+COS(X)*COS(Y)

```
>> sym.expand(sym.cos(x + y), trig=True)             Answer is -sin(x)*sin(y) + cos(x)*cos(y)
```

SIMPLIFY

(X+X*Y/X)=Y+1

```
>>sym.simplify((x + x * y) / x)                      Answer is: y+1
```

Calculus

LIMITS compute the limit of

limit(function, variable, point)

limit(sin(x)/x , x, 0) =1

Differentiation

diff(func,var) eg diff(sin(x),x)=cos(x)

diff(func,var,n) eg

Series

series(expr,var)

series(cos(x),x) = 1-x/2+x/24+o(x)

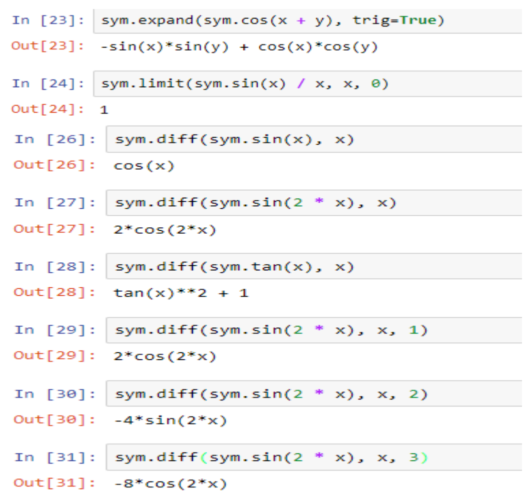
Integration

Integrate(expr,var)

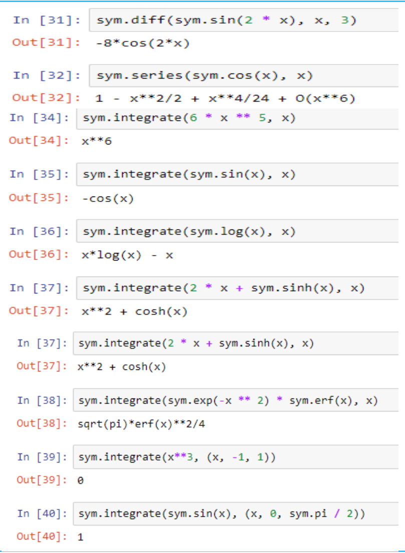
Integrate(sin(x),x) = -cos(x)

Example

Example:



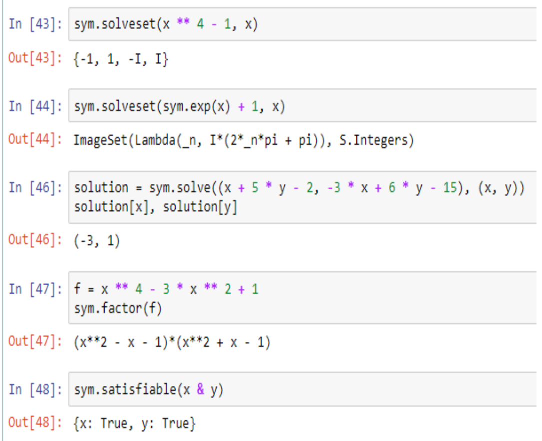
```
In [23]: sym.expand(sym.cos(x + y), trig=True)
Out[23]: -sin(x)*sin(y) + cos(x)*cos(y)

In [24]: sym.limit(sym.sin(x) / x, x, 0)
Out[24]: 1

In [26]: sym.diff(sym.sin(x), x)
Out[26]: cos(x)

In [27]: sym.diff(sym.sin(2 * x), x)
Out[27]: 2*cos(2*x)

In [28]: sym.diff(sym.tan(x), x)
Out[28]: tan(x)**2 + 1

In [29]: sym.diff(sym.sin(2 * x), x, 1)
Out[29]: 2*cos(2*x)

In [30]: sym.diff(sym.sin(2 * x), x, 2)
Out[30]: -4*sin(2*x)

In [31]: sym.diff(sym.sin(2 * x), x, 3)
Out[31]: -8*cos(2*x)
```



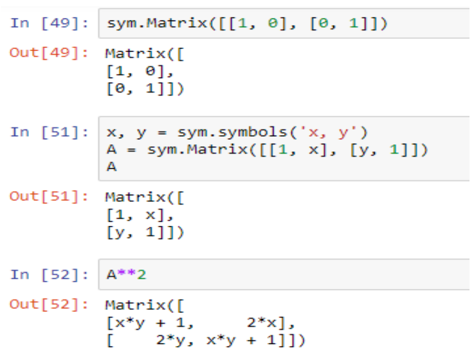
```
In [31]: sym.diff(sym.sin(2 * x), x, 3)
Out[31]: -8*cos(2*x)

In [32]: sym.series(sym.cos(x), x)
Out[32]: 1 - x**2/2 + x**4/24 + O(x**6)
In [34]: sym.integrate(6 * x ** 5, x)
Out[34]: x**6

In [35]: sym.integrate(sym.sin(x), x)
Out[35]: -cos(x)

In [36]: sym.integrate(sym.log(x), x)
Out[36]: x*log(x) - x

In [37]: sym.integrate(2 * x + sym.sinh(x), x)
Out[37]: x**2 + cosh(x)

In [37]: sym.integrate(2 * x + sym.sinh(x), x)
Out[37]: x**2 + cosh(x)

In [38]: sym.integrate(sym.exp(-x ** 2) * sym.erf(x), x)
Out[38]: sqrt(pi)*erf(x)**2/4

In [39]: sym.integrate(x**3, (x, -1, 1))
Out[39]: 0

In [40]: sym.integrate(sym.sin(x), (x, 0, sym.pi / 2))
Out[40]: 1
```

Example

Example:



```
In [43]: sym.solveset(x ** 4 - 1, x)
Out[43]: {-1, 1, -I, I}

In [44]: sym.solveset(sym.exp(x) + 1, x)
Out[44]: ImageSet(Lambda(_n, I*(2*_n*pi + pi)), S.Integers)

In [46]: solution = sym.solve((x + 5 * y - 2, -3 * x + 6 * y - 15), (x, y))
         solution[x], solution[y]
Out[46]: (-3, 1)

In [47]: f = x ** 4 - 3 * x ** 2 + 1
         sym.factor(f)
Out[47]: (x**2 - x - 1)*(x**2 + x - 1)

In [48]: sym.satisfiable(x & y)
Out[48]: {x: True, y: True}
```



```
In [49]: sym.Matrix([[1, 0], [0, 1]])
Out[49]: Matrix([
         [1, 0],
         [0, 1]])

In [51]: x, y = sym.symbols('x, y')
         A = sym.Matrix([[1, x], [y, 1]])
         A
Out[51]: Matrix([
         [1, x],
         [y, 1]])

In [52]: A**2
Out[52]: Matrix([
         [x*y + 1,     2*x],
         [    2*y, x*y + 1]])
```

Equation Solving

solveset()

solveset(x ** 4 - 1, x) ={-1,1,-I,I}

Matrices

A={[1,2][2,1]} find A**2

Automata Based Programming Paradigm

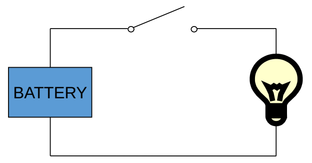
Introduction

Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other formal automation.

What is Automata Theory?

- Automata theory is the study of abstract computational devices
- Abstract devices are (simplified) models of real computations
- Computations happen everywhere: On your laptop, on your cell phone, in nature, …

Example:



input: switch

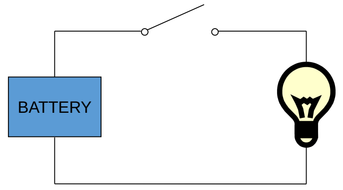
output: light bulb

actions: flip switch

states: on, off

Simple Computer

Example:



input: switch

output: light bulb

actions: flip switch

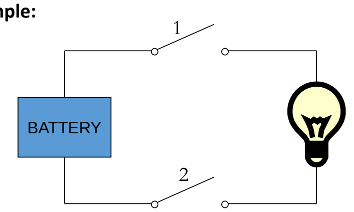
states: on, off

$$\begin{tikzcd} \text{start} \arrow[r] & \text{off} \arrow[r, bend left, "f"] & \text{on} \arrow[l, bend left, "f"] \end{tikzcd}$$

bulb is on if and only if there was an odd number of flips

Another “computer”

Example:



inputs: switches 1 and 2

actions: 1 for “flip switch 1”

actions: 2 for “flip switch 2”

states: on, off

$$\begin{tikzcd} \text{start} \arrow[r] & \text{off} \arrow[r, bend left] \arrow[d, bend left, "2"] & \text{off} \arrow[l, bend left, "1"] \arrow[d, bend left, "2"] \\ & \text{off} \arrow[u, bend left, "2"] \arrow[r, bend left, "1"] & \text{on} \arrow[u, bend left, "2"] \arrow[l, bend left, "1"] \end{tikzcd}$$

bulb is on if and only if both switches were flipped an odd number of times

Types of Automata

finite automata	Devices with a finite amount of memory. Used to model “small” computers.
push-down automata	Devices with infinite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.

Alphabets and strings

A common way to talk about words, number, pairs of words, etc. is by representing them as strings

To define strings, we start with an alphabet

An alphabet is a finite set of symbols.

Examples:

$\Sigma_1 = \{a, b, c, d, \ldots, z\}$: the set of letters in English

$\Sigma_2 = \{0, 1, \ldots, 9\}$: the set of (base 10) digits

$\Sigma_3 = \{a, b, \ldots, z, \#\}$: the set of letters plus the special symbol #

$\Sigma_4 = \{\, (, \,) \,\}$: the set of open and closed brackets

Strings

A string over alphabet Σ is a finite sequence of symbols in Σ.

The empty string will be denoted by e

Examples:

$abfbz$ is a string over $\Sigma_1 = \{a, b, c, d, \ldots, z\}$

9021 is a string over $\Sigma_2 = \{0, 1, \ldots, 9\}$

$ab\#bc$ is a string over $\Sigma_3 = \{a, b, \ldots, z, \#\}$

$))()(()$ is a string over $\Sigma_4 = \{\, (, \,) \,\}$

Languages

A language is a set of strings over an alphabet.

Languages can be used to describe problems with "yes/no" answers, for example:

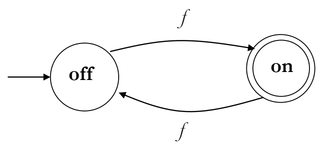
$L_1 =$ The set of all strings over Σ_1 that contain the substring "SRM"

$L_2 =$ The set of all strings over Σ_2 that are divisible by 7 $= \{7, 14, 21, \ldots\}$

$L_3 =$ The set of all strings of the form $s\#s$ where s is any string over $\{a, b, \ldots, z\}$

$L_4 =$ The set of all strings over Σ_4 where every (can be matched with a subsequent)

Finite Automata



There are states off and on, the automaton starts in off and tries to reach the "good state" on

What sequences of fs lead to the good state?

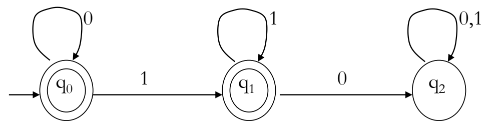
Answer: {f, fff, fffff, …} = {f n: n is odd}

This is an example of a deterministic finite automaton over alphabet {f}

Deterministic finite automata

- A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
  - Q is a finite set of states
  - Σ is an alphabet
  - $\delta: Q \times \Sigma \to Q$ is a transition function
  - $q_0 \in Q$ is the initial state
  - $F \subseteq Q$ is a set of accepting states (or final states).
- In diagrams, the accepting states will be denoted by double loops

Example



alphabet $\Sigma = \{0, 1\}$
start state $Q = \{q_0, q_1, q_2\}$
initial state q_0
accepting states $F = \{q_0, q_1\}$

transition function δ:

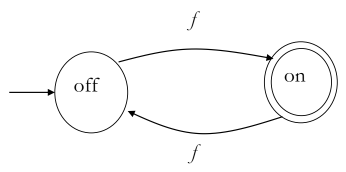
inputs

states	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

Language of a DFA

The language of a DFA $(Q, \Sigma, \delta, q_0, F)$ is the set of all strings over Σ that, starting from q_0 and following the transitions as the string is read left to right, will reach some accepting state.

M:

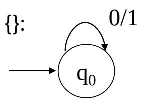


- Language of M is $\{f, fff, fffff, \ldots\} = \{f^n: n \text{ is odd}\}$

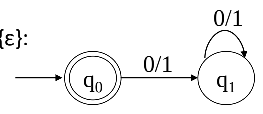
Example of DFA

1. Let Σ = {0, 1}. Give DFAs for {}, {ε}, Σ*, and Σ+.

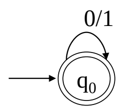
For {}:



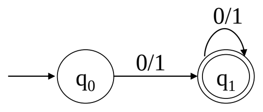
For {ε}:



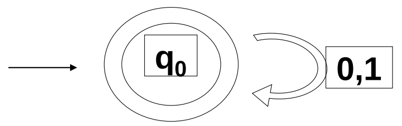
For Σ*:



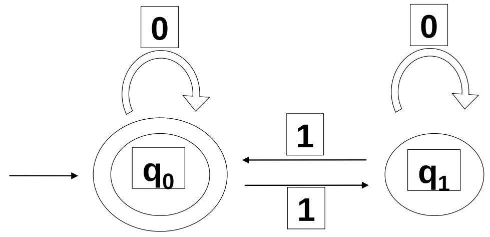
For Σ+:



Example of DFA



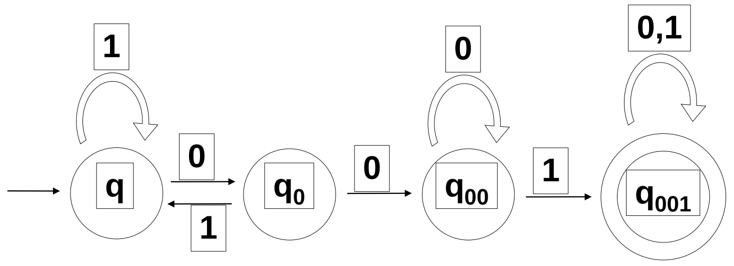
L(M) = {0,1}*



L(M) = { w | w has an even number of 1s}

Example of DFA

Build an automaton that accepts all and only those strings that contain 001



Example of DFA using Python

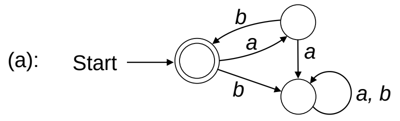
```
from automata.fa.dfa import DFA
# DFA which matches all binary strings ending in an odd number of '1's
dfa = DFA(
   states={'q0', 'q1', 'q2'},
   input_symbols={'0', '1'},
   transitions={
      'q0': {'0': 'q0', '1': 'q1'},
      'q1': {'0': 'q0', '1': 'q2'},
      'q2': {'0': 'q2', '1': 'q1'}
   },
   initial_state='q0',
   final_states={'q1'}
)
dfa.read_input('01')   # answer is  'q1‘
dfa.read_input('011')  # answer is error
print(dfa.read_input_stepwise('011'))
Answer # yields:
# 'q0‘     # 'q0‘     # 'q1'
# 'q2‘     # 'q1'
```

```
if dfa.accepts_input('011'):
   print('accepted')
else:
   print('rejected')
```

Questions for DFA

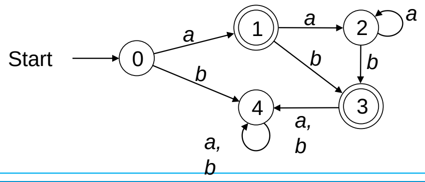
Find an DFA for each of the following languages over the alphabet $\{a, b\}$.
(a) $\{(ab)^n \mid n \in \mathbb{N}\}$, which has regular expression $(ab)^*$.

Solution:



b) Find a DFA for the language of $a + aa^*b$.

Solution:



Questions for DFA

c) A DFA that accepts all strings that contain 010 or do not contain 0.

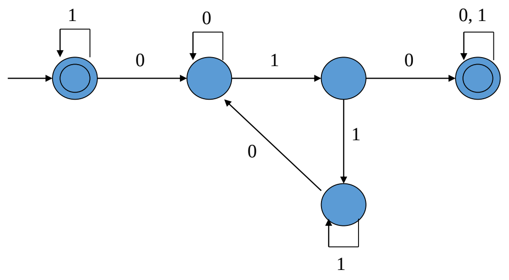
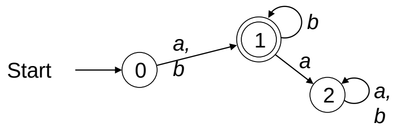


Table Representation of a DFA

A DFA over A can be represented by a transition function T : States X A -> States, where T(i, a) is the state reached from state i along the edge labelled a, and we mark the start and final states. For example, the following figures show a DFA and its transition table.



	T	a	b
start	0	1	1
final	1	2	1
	2	2	2

Sample Exercises - DFA

1. Write a automata code for the Language that accepts all and only those strings that contain 001
2. Write a automata code for L(M) ={ w | w has an even number of 1s}
3. Write a automata code for L(M) ={0,1}*
4. Write a automata code for L(M)=*a* + *aa***b*.
5. Write a automata code for L(M)={$(ab)^n$ | $n \in$ N}
6. Write a automata code for Let Σ = {0, 1}.

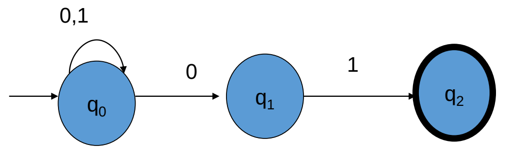
   Given DFAs for {}, {ε}, Σ^*, and Σ^+.

NDFA

- A nondeterministic finite automaton M is a five-tuple M = (Q, Σ, δ, q_0, F), where:
  - Q is a finite set of states of M
  - Σ is the finite input alphabet of M
  - δ: Q × Σ → power set of Q, is the state transition function mapping a state-symbol pair to a subset of Q
  - q_0 is the start state of M
  - F ⊆ Q is the set of accepting states or final states of M

Example NDFA

- NFA that recognizes the language of strings that end in 01



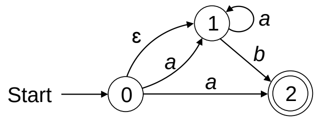
Exercise: Draw the complete transition table for this NFA

note: $\delta(q_0,0) = \{q_0,q_1\}$
$\delta(q_1,0) = \{\}$

NDFA

A nondeterministic finite automaton (NFA) over an alphabet A is similar to a DFA except that epislon-edges are allowed, there is no requirement to emit edges from a state, and multiple edges with the same letter can be emitted from a state.

Example. The following NFA recognizes the language of a + aa*b + a*b.

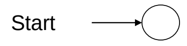


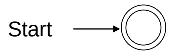
	T	a	b	e
start	0	{1, 2}	∅	{1}
	1	{1}	{2}	∅
final	2	∅	∅	∅

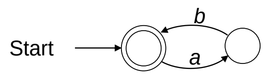
Table representation of NFA

An NFA over A can be represented by a function T : States × A ∪ {L} → power(States), where T(i, a) is the set of states reached from state i along the edge labeled a, and we mark the start and final states. The following figure shows the table for the preceding NFA.

Examples

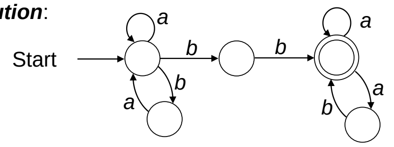
Solutions: (a): 

(b): 

(c): 

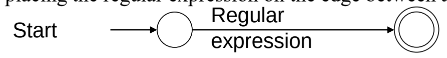
Find an NFA to recognize the language (*a* + *ba*)**bb*(*a* + *ab*)*.

A solution:

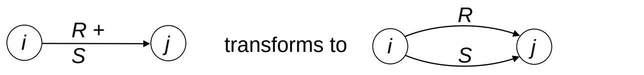


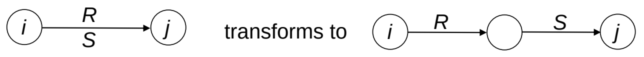
Examples

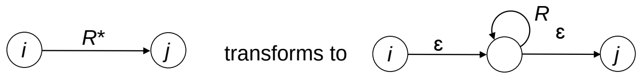
Algorithm: *Transform a Regular Expression into a Finite Automaton*
Start by placing the regular expression on the edge between a start and final state:



Apply the following rules to obtain a finite automaton after erasing any ∅-edges.

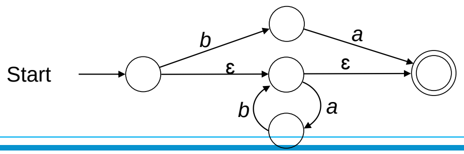






Quiz. Use the algorithm to construct a finite automaton for (*ab*)* + *ba*.

Answer:



Example of NFA using Python

```
from automata.fa.nfa import NFA
# NFA which matches strings beginning with 'a', ending with 'a', and containing
# no consecutive 'b's
nfa = NFA(
   states={'q0', 'q1', 'q2'},
   input_symbols={'a', 'b'},
   transitions={
      'q0': {'a': {'q1'}},
      # Use '' as the key name for empty string (lambda/epsilon) transitions
      'q1': {'a': {'q1'}, '': {'q2'}},
      'q2': {'b': {'q0'}}
   },
   initial_state='q0',
   final_states={'q1'}
)
```

```
nfa.read_input('aba')
ANSWER :{'q1', 'q2'}

nfa.read_input('abba')
ANSWER: ERROR


nfa.read_input_stepwise('aba')

if nfa.accepts_input('aba'):
   print('accepted')
else:
   print('rejected')
ANSWER: ACCEPTED
nfa.validate()
ANSWR: TRUE
```

Sample Exercises - NFA

1. Write a automata code for the Language that accepts all end with 01
2. Write a automata code for L(M)= $a + aa^*b + a^*b$.
3. Write a automata code for Let $\Sigma = \{0, 1\}$.

Given NFAs for {}, {ε}, $\{(ab)^n \mid n \in \mathbb{N}\}$, which has regular expression $(ab)^*$.

Dependent type Programming Paradigm

Introduction

A constant problem:

- Writing a correct computer program is hard and proving that a program is correct is even harder
- Dependent Types allow us to write programs and know they are correct before running them.
- dependent types: you can specify types that can check the value of your variables at compile time

Example:

Here is how you can declare a Vector that contains the values 1, 2, 3 :

```
val l1 = 1 :#: 2 :#: 3 :#: Vnil
```

This creates a variable l1 who's type signature specifies not only that it's a Vector that contains Ints, but also that it is a Vector of length 3. The compiler can use this information to catch errors. Let's use the vAdd method in Vector to perform a pairwise addition between two Vectors:

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
val l2 = 1 :#: 2 :#: 3 :#: VNil
 val l3 = l1 vAdd l2
 // Result: l3 = 2 :#: 4 :#: 6 :#: VNil
```

Introduction

The example above works fine because the type system knows both Vectors have length 3. However, if we tried to vAdd two Vectors of different lengths, we'd get an error at compile time instead of having to wait until run time!

```
        val l1 = 1 :#: 2 :#: 3 :#: VNil
        val l2 = 1 :#: 2 :#: VNil

        val l3 = l1 vAdd l2

// Result: a *compile* error because you can't pairwise add vectors
// of different lengths!
```

Note:

You can express almost anything with dependent types. A factorial function which only accepts natural numbers, a login function which doesn't accept empty strings, a remove Last function which only accepts non-empty arrays. And all this is checked before you run the program.

Introduction

A function has dependent type if the type of a function's result depends on the VALUE of its argument; this is not the same thing as a ParameterizedType. The second order lambda calculus possesses functions with dependent types.

What does it mean to be "correct"?

Depends on the application domain, but could mean one or more of:

- Functionally correct (e.g. arithmetic operations on a CPU)
- Resource safe (e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . .)
- Secure (e.g. not allowing access to another user's data)

What is type?

- In programming, types are a means of classifying values
- Exp: values 94, "thing", and [1,2,3,4,5] classified as an integer, a string, and a list of integers
- For a machine, types describe how bit patterns in memory are to be interpreted.
- For a compiler or interpreter, types help ensure that bit patterns are interpreted consistently when a program runs.
- For a programmer, types help name and organize concepts, aiding documentation and supporting interactive editing environments.

Introduction

In computer science and logic, a dependent type is a type whose definition depends on a value.

It is an overlapping feature of type theory and type systems.

Used to encode logic's quantifiers like "for all" and "there exists".

Dependent types may help reduce bugs by enabling the programmer to assign types that further restrain the set of possible implementations.

Quantifiers

A predicate becomes a proposition when we assign it fixed values. However, another way to make a predicate into a proposition is to quantify it. That is, the predicate is true (or false) for all possible values in the universe of discourse or for some value(s) in the universe of discourse. Such quantification can be done with two quantifiers: the universal quantifier and the existential quantifier.

Universal: Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing. The universal quantification of a predicate P(x) is the proposition "P(x) is true for all values of x in the universe of discourse" We use the notation ∀ for Universal quantifier

$\forall x P(x)$

which can be read "for all x"

Example:

Let P(x) be the predicate "x must take a discrete mathematics course" and let Q(x) be the predicate "x is a computer science student". The universe of discourse for both P(x) and Q(x) is all UNL students.

Express the statement "Every computer science student must take a discrete mathematics course".

$\forall x(Q(x) \rightarrow P(x))$

Express the statement "Everybody must take a discrete mathematics course or be a computer science student".

$\forall x(Q(x) \vee P(x))$

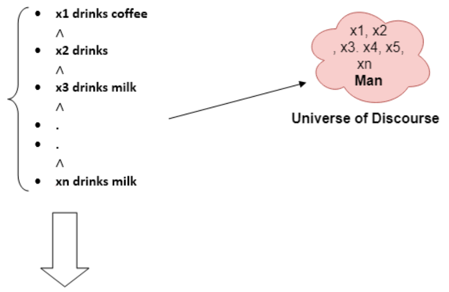
If x is a variable, then ∀x is read as:

For all x **For each x** **For every x.**

Quantifiers

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



So in shorthand notation, we can write it as :

∀x man(x) → drink (x, coffee).
It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifiers

The existential quantification of a predicate P(x) is the proposition “There exists an x in the universe of discourse such that P(x) is true.” We use the notation

∃xP(x)

which can be read “there exists an x”

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

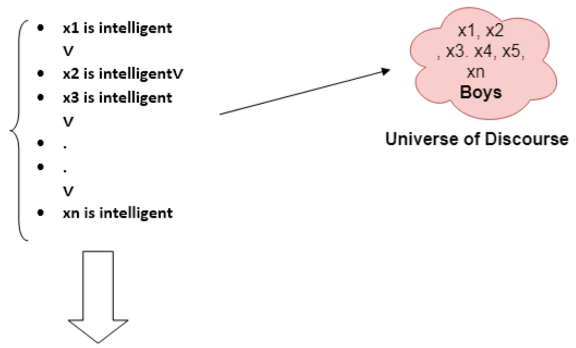
There exists a 'x.'

For some 'x.'

For at least one 'x.'

Existential Quantifiers

Some boys are intelligent.



So in short-hand notation, we can write it as:

∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Examples

1. All birds fly.

In this question the predicate is "fly(bird)."

And since there are all birds who fly so it will be represented as follows.

∀x bird(x) →fly(x).

2. Every man respects his parent.

In this question, the predicate is "respect(x, y)," where x=man, and y= parent.

Since there is every man so will use ∀, and it will be represented as follows:

∀x man(x) → respects (x, parent).

3. Some boys play cricket.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use ∃, and it will be represented as:

∃x boys(x) → play(x, cricket).