Academic Year:2023-24 SAP ID:

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL405 **DATE: 7 February 2024**

COURSE NAME: Programing Laboratory 2 (Python) **CLASS:** S.Y. B.Tech

NAME: Anish Sharma **SAP ID:** 60003220045

EXPERIMENT NO. 1

CO/LO: CO1, CO2.

AIM / OBJECTIVE:

Write python programs to understand

(a)Expressions, Variables.

(b)Quotes, Basic Math operations.

(c) Basic String Operations & String Methods.

DESCRIPTION OF EXPERIMENT:

1. Variables, expressions and statements

1.1. Values and data types

A **value** is one of the fundamental things — like a letter or a number — that a program manipulates. The values we have seen so far are 2 (the result when we added 1 + 1), and "Hello, World!".

These values belong to different **data types**: 2 is an *integer*, and "Hello, World!" is a *string*, so-called because it contains a string of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

```
>>> print 4
4
```

If you are not sure what type a value has, the interpreter can tell you.

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

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

Not surprisingly, strings belong to the type **str** and integers belong to the type **int**. Less obviously, numbers with a decimal point belong to a type called **float**, because these numbers are represented in a format called *floating-point*.

```
>>> type(3.2)
<type 'float'>
```

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

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

They're strings.

Strings in Python can be enclosed in either single quotes (') or double quotes ("):

```
>>> type('This is a string.')
<type 'str'>
>>> type("And so is this.")
<type 'str'>
```

Double quoted strings can contain single quotes inside them, as in "Bruce's beard", and single quoted strings can have double quotes inside them, as in 'The knights who say "Ni!"'.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> print 1,000,000
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a list of three items to be printed. So remember not to put commas in your integers.

1.2. Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

The **assignment statement** creates new variables and gives them values:

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

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second gives the integer 17 to n, and the third gives the floating-point number 3.14159 to pi.

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

The type of a variable is the type of the value it refers to.

1.3. Variable names and keywords

Programmers generally choose names for their variables that are meaningful — they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. Bruce and bruce are different variables.

The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or price_of_tea_in_china.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what's wrong with class?

It turns out that class is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names.

Python has thirty-one keywords:

and	as	assert	break	class	continue

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	not	or	pass
print	raise	return	try	while	with
yield					

1.4. Statements

A **statement** is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

1.5. Evaluating expressions

An **expression** is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
```

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

2 Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.

There are 7 arithmetic operators in Python :

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulus

6. Exponentiation
7. Floor division

Python also provides three more operators for us to use:

1. **, for exponentiation.

2. //, for integer division

3. %, for modulus

The following sections describe exactly how each of these operators work, and the order of operations for each of them.

Exponentiation (**)

Following are examples of its use:

```
>>> 2 ** 3

8

>>> 3 ** 5

>>> 25 ** .5
5.0
>>> 2 ** -3
0.125
 >>> 9999 ** 0
1
 >>> -5 ** -3
-0.008
>>> -5 ** 3 -125
>>> 1.6743 ** 2.3233
3.311554089370817
```

Integer division (/ /)

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

Python provides a second division operator, //, which performs integer division. In particular, the answer of an integer division operation is always an integer. In particular, a//b is defined as the largest number of whole times b divides into a. Here are a few examples of evaluating expressions with integer division:

```
>>> 25//4
6
 >>> 13//6
 2
>>> 100//4
25
```

Modulus Operator (%)

In python, the modulus operator is defined for both integers and real numbers. If both operands are integers, then the answer will be an integer, otherwise, it will be a real number.

The formal definition of a % b is as follows: a % b evaluates to a - (a // b)*b. a // b represents the whole number of times b divides into a.

Thus, we are looking for the total number of times b goes into a, and subtracting out of a, that many multiples of b, leaving the "leftover."

Here are some conventional examples of mod, using only non-negative numbers:

```
>>> 17 % 3
 2
 >>> 37 % 4
1
>>> 17 % 9
8
```

2.1. Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:

1. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.
2. **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.
3. **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).
4. Operators with the same precedence are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59*1, which is 59, which is wrong.

2.2. Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```
message-1   "Hello"/123   message*"Hello"   "15"+2
```

Interestingly, the + operator does work with strings, although it does not do exactly what you might expect. For strings, the + operator represents **concatenation**, which means joining the two operands by linking them end-to-end. For example:

```
fruit = "banana"
baked_good = " nut bread"
print fruit + baked_good
```

The output of this program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.

The * operator also works on strings; it performs repetition. For example, 'Fun'*3 is 'FunFunFun'. One of the operands has to be a string; the other has to be an integer.

On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect "Fun"*3 to be the same as "Fun"+"Fun"+"Fun", and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

String Manipulation in Python

Strings are sequences of characters. Python strings are "**immutable**" which means they cannot be changed after they are created. To create a string, put the sequence of characters inside either single quotes, double quotes, or triple quotes and then assign it to a variable.

Creating strings is as simple as assigning a value to a variable. For example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

```
var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

When the above code is executed, it produces the following result −

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

```
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

When the above code is executed, it produces the following result −

```
Updated String :-  Hello Python
```

String Methods

Python has several built-in methods associated with the string data type. These methods let us easily modify and manipulate strings. Built-in methods are those that are defined in the Python

programming language and are readily available for us to use. Here are some of the most common string methods.

Method-Description(not limited to)

1. capitalize()-Converts the first character to upper case
2. casefold()-Converts string into lower case
3. center()-Returns a centered string
4. count()-Returns the number of times a specified value occurs in a string
5. encode()-Returns an encoded version of the string
6. endswith()-Returns true if the string ends with the specified value
7. expandtabs()-Sets the tab size of the string
8. find()-Searches the string for a specified value and returns the position of where it was found
9. format()-Formats specified values in a string
10. format_map()-Formats specified values in a string
11. index()-Searches the string for a specified value and returns the position of where it was found
12. isalnum()-Returns True if all characters in the string are alphanumeric
13. isalpha()-Returns True if all characters in the string are in the alphabet
14. isdecimal()-Returns True if all characters in the string are decimals
15. isdigit()-Returns True if all characters in the string are digits
16. islower()-Returns True if all characters in the string are lower case
17. isnumeric()-Returns True if all characters in the string are numeric
18. isprintable()-Returns True if all characters in the string are printable
19. isspace()-Returns True if all characters in the string are whitespaces
20. istitle()-Returns True if the string follows the rules of a title
21. isupper()-Returns True if all characters in the string are upper case
22. join()-Converts the elements of an iterable into a string
23. ljust()-Returns a left justified version of the string
24. lower()-Converts a string into lower case
25. lstrip()-Returns a left trim version of the string
26. maketrans()-Returns a translation table to be used in translations
27. partition()-Returns a tuple where the string is parted into three parts
28. replace()-Returns a string where a specified value is replaced with a specified value
29. rfind()-Searches the string for a specified value and returns the last position of where it was found
30. rindex()-Searches the string for a specified value and returns the last position of where it was found

31. rjust()-Returns a right justified version of the string
32. rpartition()-Returns a tuple where the string is parted into three parts
33. rsplit()-Splits the string at the specified separator, and returns a list
34. rstrip()-Returns a right trim version of the string
35. split()-Splits the string at the specified separator, and returns a list
36. splitlines()-Splits the string at line breaks and returns a list
37. startswith()-Returns true if the string starts with the specified value
38. strip() Returns a trimmed version of the string
39. swapcase()-Swaps cases, lower case becomes upper case and vice versa
40. title()-Converts the first character of each word to upper case
41. translate() Returns a translated string
42. upper()-Converts a string into upper case
43. zfill()-Fills the string with a specified number of 0 values at the beginning

3.2 Input

There are two built-in functions in Python for getting keyboard input:

```
n = raw_input("Please enter your name: ")
print n
n = input("Enter a numerical expression: ")
print n
```

A sample run of this script would look something like this:

```
$ python tryinput.py
Please enter your name: Arthur, King of the Britons
Arthur, King of the Britons
Enter a numerical expression: 7 * 3
21
```

Each of these functions allows a *prompt* to be given to the function between the parentheses.

3.3 Composition

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

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>> print 17 + 3
20
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

In reality, the addition has to happen before the printing, so the actions aren't actually happening at the same time. The point is that any expression involving numbers, strings, and variables can be used inside a print statement. You've already seen an example of this:

```
print "Number of minutes since midnight: ", hour*60+minute
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
percentage = (minute * 100) / 60
```

This ability may not seem impressive now, but you will see other examples where composition makes it possible to express complex computations neatly and concisely.

Warning: There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a *variable* name, not an expression. So, the following is illegal: `minute+1 = hour`.

3.4 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they are marked with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # caution: integer division
```

Everything from the # to the end of the line is ignored — it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code. In this case, it reminds the reader about the ever-surprising behavior of integer division.

Arithmetic Expressions in Python

Python has its set of rules about how these expressions are to be evaluated, so that there is no ambiguity.

Multiplication and division have higher precedence than addition and subtraction, as is frequently taught in grade school mathematics. Furthermore, parentheses have the highest precedence and can be used to "force"

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

the order in which operations are evaluated as is illustrated in the this line of code that was previously shown:

total_price = item_price*(1+tax_rate/100)

In this expression, we first evaluate the contents of the parentheses before multiplying. In evaluating the contents of the parentheses, we first perform the division, since that has higher precedence than addition. Thus, for example, if tax_rate is 7, then we first take 7/100 to get .07 and then add that to 1 to get 1.07. Then, the current value of item_price is multiplied by 1.07, which is then assigned to total_price.

Input statement

Python makes reading input from the user very easy. In particular, Python makes sure that there is always a prompt (a print) for the user to enter some information.

Consider the following example entered

```
>>> name = input("What is your name?\n")
What is your name?
Simone
>>> print("Please to meet you ", name, ".", sep="")
Please to meet you Simone.
age = int(input("How old are you?\n")) if 15 < age < 25: print("You may drive, but not rent a car.")
```

QUESTIONS:

1. Write a python program to take the input from the user for the first name and last name and concatenate both the strings. Also add comments to the program

2. Write a program to evaluate the polynomial shown here:

   $3x^3 - 5x^2 + 6$ for x = 2.55.

3. Write a program to output middle three characters of an input string.
4. Arrange string characters such that lowercase letters should come first.

   For example: str1='PyTHon" then output should be "yonPTH"

5. Count all letters, digits, and special symbols from a given string.
6. Write a program to count occurrences of all characters within a string.
7. Write a program to find the last position of a substring "**Rama**" in a given string

   For e.g. "Mary always stood first in class. Mary now works at Google."

The expected outcome is "The last position of Mary starts at index 34"

8. Removal all characters from a string except integers
9. Replace each special symbol with # in the following string:

   I/p string:Mary @always &stood firl̥st in %class
   O/P String: Mary #always #stood first in #class

10. Write a program that takes a sentence as an input parameter where each word in the sentence is separated by a space. Then replace each blank with a hyphen and then print the modified sentence.

Questions for Write-up:

1. **What is Python? What are the benefits of using Python**
2. **List and explain various string functions used in programs executed.**
3. **Explain the difference between a List, Tuple and Set.**

OBSERVATIONS / DISCUSSION OF RESULT:

This section should interpret the outcome of the experiment. The observations can be visually represented using images, tables, graphs, etc. This section should answer the question "What do the result tell us?" Compare and interpret your results with expected behavior. Explain unexpected behavior, if any.

CONCLUSION:

Base all conclusions on your actual results; describe the meaning of the experiment and the implications of your results.

REFERENCES:

Website References:

1. https://www.w3schools.com/python
2. https://www.tutorialspoint.com/
3. https://www.programiz.com

Name: Anish Sharma

SAP ID: 60003220045

Roll no: I011

Experiment no: 1

IT1

1. Write a python program to take the input from the user for the first name and last name and concatenate both the strings. Also add comments to the program

```
first = input("ENter first name:")
last = input("Enter last name:")
```

```
ENter first name:Anish
Enter last name:Sharma
```

```
full_name =first+last
full_name
```

```
'AnishSharma'
```

1. Write a program to evaluate the polynomial shown here:

3x3 - 5x2 + 6 for x = 2.55

```
3*pow(2.55,3)-5*pow(2.55,2)+6
```

```
23.231625
```

1. Write a program to output middle three characters of an input string.

```
x=input("String:")
```

```
String:Anish
```

```
mid=len(x)//2
print(x[mid-1:mid+2])
```

```
nis
```

1. Arrange string characters such that lowercase letters should come first.

For example: str1='PyTHon" then output should be "yonPTH"

```
```

```
str1='PyTHon'
y=""
for i in str1:
    x=i
    if x.islower():
        y+=x

for i in str1:
    x=i
    if x.isupper():
        y+=x
print(y)
```

```
yonPTH
```

1. Count all letters, digits, and special symbols from a given string.

```
s = "Anish@123"
digit=0
alpha=0
special=0
for i in s:
    if i.isdigit():
        digit+=1
    elif i.isalpha():
        alpha+=1
    else:
        special+=1


digit
```

```
3
```

```
special
```

```
1
```

```
alpha
```

```
5
```

1. Write a program to count occurrences of all characters within a string.

```
s = "aaababcccddac"

unique = set(s)
unique
```

```
{'a', 'b', 'c', 'd'}
```

```
for i in unique:
    ctr=0
    for j in s:
        if i==j:
            ctr+=1
    print("occurrence of ",i," is ",ctr)
```

```
occurrence of  d  is  2
occurrence of  c  is  4
occurrence of  b  is  2
occurrence of  a  is  5
```

1. Write a program to find the last position of a substring “Rama” in a given string

For e.g. "Mary always stood first in class. Mary now works at Google."

The expected outcome is “The last position of Mary starts at index 34”

```
s="Mary always stood first in class. Mary now works at Google."
s.rfind('Mary')
```

```
34
```

1. Removal all characters from a string except integers

```
s = 'Anis67h123456'
x=''
for i in s:
    if i.isdigit():
        x+=i
s=x
s
```

```
'67123456'
```

9.Replace each special symbol with # in the following string:

I/p string:Mary @always &stood fir!st in %class

O/P String: Mary #always #stood first in #class

```
import string

a = "Mary @always &stood fir!st in %class"
for i in string.punctuation:
    a=a.replace(i,"#")
print(a)
```

```
Mary #always #stood fir#st in #class
```

1. Write a program that takes a sentence as an input parameter where each word in the sentence is separated by a space. Then replace each blank with a hyphen and then print the modified sentence.

```
a = input("String:")
s =''
for i in a:
    if i!=' ':
       s+=i
    else:
        s+='-'
s
```

```
String:jv fjg
```

```
'jv-fjg'
```