PROCEDURAL - Conditions

- Python Decision Making
  - if statements
  - if...else statements
  - elif Statement
  - nested if statements
  - Single Statement Suites

if Statement

- **if test expression:**
- **statement(s)**
  - The program evaluates the test expression and will execute statement(s) only if the text expression is True.
  - If the text expression is False, the statement(s) is not executed.
  - The body of the if statement is indicated by the indentation.
    - Body starts with an indentation and the first unindented line marks the end.
  - Python interprets non-zero values as True.
  - None and 0 are interpreted as False.
- Example

Program	**Output**
x = int (input("value-1:"))	
print(x)	
y = int (input("value-2:"))	value-1:4
print(y)	4
if x<y:	value-2:5
print("The first value is smaller thus y-x is ", (y-x))	5
x = int (input("value-1:"))	The first value is smaller thus y-x is 1

if … else Statement

- **if test expression:**
- **Body of if**
- **else:**
- **Body of else**
  - The if..else statement evaluates test expression and will execute body of if only when test condition is True.
  - If the condition is False, body of else is executed.
  - Indentation is used to separate the blocks.

- Example

Program	Output
num=int(input("Enter the number:"))	
if num >= 0:	Enter the number:4
print("Positive or Zero")	Positive or Zero
else:	Enter the number:-2
print("Negative number")	Negative number

elif Statement

- The elif statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.
- elif statement is optional.
- There can be an arbitrary number of elif statements following an if.
  - **if expression1:**
  -         **statement(s)**
  - **elif expression2:**
  -         **statement(s)**
  - **elif expression3:**
  -         **statement(s)**
  - **else:**
  -         **statement(s)**
- Example

Program	Output
a=int(input("Enter the number a:"))	Enter the number a:3
b=int(input("Enter the number b:"))	Enter the number b:1
if b > a:	
print("b is greater than a")	Enter the number a:2
elif a == b:	Enter the number b:3
print("a and b are equal")	b is greater than a

Nested if statement

- A if...elif...else statement can be written inside another if...elif...else statement.
  - This is called nesting in computer programming.
- Any number of these statements can be nested inside one another.
- Indentation is the only way to figure out the level of nesting.

- Example

Program	Output
num = float(input("Enter a number: "))	
if num >= 0:	
if num == 0:	Enter a number: 10
print("Zero")	Positive number
else:	Enter a number: 0
print("Positive number")	Zero
else:	Enter a number: -9
print("Negative number")	Negative number

Program	Output
marks = int(input("Enter marks"))	Enter marks89
print (marks)	89
if marks >= 40:	Student is pass
print ("Student is pass")	Grade 'A'
if marks >= 90:	Enter marks30
print ("Grade 'S'")	30
elif marks >=80 and marks <90:	Student is fail
print ("Grade 'A'")	Enter marks67
elif marks >=70 and marks <80:	67
print ("Grade 'B'")	Student is pass
elif marks >=60 and marks <70:	Grade 'C'
print ("Grade 'C'")	
elif marks >=50 and marks <60:	
print ("Grade 'D'")	
elif marks >=40 and marks <50:	
print ("Grade 'E'")	
else:	
print("Student is fail")	

Conditions

- If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.

Loops

- Python uses two loops
  - while
  - for

while

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
  - **while test-expression:**
  - **statement**
    - In while loop, test expression is checked first.
    - The body of the loop is entered only if the test_expression evaluates to True.
    - After one iteration, the test expression is checked again.
    - This process continues until the test_expression evaluates to False.
- In Python, the body of the while loop is determined through indentation.
- Body starts with indentation and the first unindented line marks the end.
- Python interprets any non-zero value as True.
  - None and 0 are interpreted as False.
- while can have an optional else block.
- The else part is executed if the condition in the while loop evaluates to False.
  - **while test_expression:**
  - **body**
  - **else:**
  - **statement**
- Example

Program	Output
n = int(input("Enter n: "))	
sum = 0	
i = 0	
if n < 0 :	
while i >= n:	
sum = sum + i	
i = i - 1 # update counter	
else:	
while i <= n:	Enter n: 5
sum = sum + i	The sum is 15
i = i+1 # update counter	Enter n: -5
print("The sum is", sum)	The sum is -15

Program	Output
counter = 0	
while counter < 3:	
print("Inside loop with counter = ", counter)	Inside loop with counter = 0
counter = counter + 1	Inside loop with counter = 1
else:	Inside loop with counter = 2
print("Inside else with counter = ", counter)	Inside else with counter = 3

for

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.
- Iterating over a sequence is called traversal.
  - **for val in sequence:**
  -         **Body of for**
    - val is the variable that takes the value of the item inside the sequence on each iteration.
- Loop continues until the last item in the sequence is reached.
- The body of for loop is separated from the rest of the code using indentation.
- Example

Program	Output
print("Enter the numbers separated by ,")	
numbers = [int(x) for x in input().split(',')]	
#numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]	
sum = 0	
for val in numbers:	Enter the numbers separated by ,
sum = sum+val	2,3,4,5,6,7,9
print("The sum is", sum)	The sum is 36

range

- A sequence of numbers can be generated using range() function.
  - range(10) will generate numbers from 0 to 9 (10 numbers).
- The start, stop and step size can be defined as **range(start, stop, step size)**.
  - step size defaults to 1 if not provided.
- This function does not store all the values in memory, it would be inefficient.
  - It remembers the start, stop, step size and generates the next number on the go.

- To force this function to output all the items, use it as an argument to the list() constructor.

for with range

- range() can be used with for loops to iterate through a sequence of numbers.
  - **for var in range():**
  - **statements using var as index**
- Example

Program	Output
genre = ['pop', 'rock', 'jazz', 'classical', 'EDM']	I like pop
	I like rock
# iterate over the list using index	I like jazz
for i in range(len(genre)):	I like classical
print("I like", genre[i])	I like EDM

for with else

- A for loop can have an optional else block as well.
- The else part is executed if the items in the sequence used in for loop exhausts.
  - **for val in sequence:**
  - **Body of for**
  - **else:**
  - **statement**
- Example

Program	Output
digits = [0, 1, 5]	
for i in digits:	0
print(i)	1
else:	5
print("No items left.")	No items left.

break

- The break statement terminates the loop containing it.
- Control of the program flows to the statement immediately after the body of the loop.
- If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.
  - **Break**
- Example

Program	Output
for val in "string":	
if val == "i":	s
break	t
print(val)	r
print("The end")	The end

continue

- The continue statement is used to skip the rest of the code inside a loop for the current iteration only.
- Loop does not terminate but continues on with the next iteration.
  - **Continue**
- Example

Program	Output
for val in "string":	s
if val == "i":	t
continue	r
print(val)	n
print("The end")	g
	The end

Format()

- The format() method takes two parameters:
  - value - value that needs to be formatted
  - format_spec - The specification on how the value should be formatted.
- The format specifier could be in the format:
  - **[[fill]align][sign][#][0][width][,][.precision][type]**
    - where, the options are
      - fill - any character
      - align - "<" | ">" | "=" | "^"
      - sign - "+" | "-" | " "
      - width - integer
      - precision - integer
      - type - "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%

Type

Type	Meaning
d	Decimal integer
c	Corresponding Unicode character
b	Binary format
O	Octal format
X	Hexadecimal format (lower case)
X	Hexadecimal format (upper case)
N	Same as 'd'. Except it uses current locale setting for number separator
E	Exponential notation. (lowercase e)
E	Exponential notation (uppercase E)
f	Displays fixed point number (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
%	Percentage. Multiples by 100 and puts % at the end.

Alignment

Type	Meaning
<	Left aligned to the remaining space
^	Center aligned to the remaining space
>	Right aligned to the remaining space
=	Forces the signed (+) (-) to the leftmost position

Format in print

- ₪ To display with space in the same line
    - φ print(var, end=" ")
- ₪ To change the order of list elements
    - φ print('{1} {2} {0}'.format('one', 'two', 'three'))
- ₪ Example

Statement	Output
print(format(123, "d"))	123

print(format(123.4567898, "f"))	123.456790
print(format(12, "b"))	1100
print(format(1234, "*>+7,d"))	*+1,234
print(format(123.4567, "^-09.3f"))	0123.4570
print("The number is:{:d}".format(123))	The number is:123
print("The float number is:{:f}".format(123.4567898))	The float number is:123.456790
print("bin: {0:b}, oct: {0:o}, hex: {0:x}".format(12))	bin: 1100, oct: 14, hex: c
print("{:5d}".format(12))	12
print("{:2d}".format(1234))	1234
print("{:8.3f}".format(12.2346))	12.235
print("{:05d}".format(12))	00012
print("{:08.3f}".format(12.2346))	0012.235
print("{:+f} {:+f}".format(12.23, -12.23))	+12.230000 -12.230000
print("{:-f} {:-f}".format(12.23, -12.23))	12.230000 -12.230000
print("{: f} {: f}".format(12.23, -12.23))	12.230000 -12.230000
print("{:5d}".format(12))	12
print("{:^10.3f}".format(12.2346))	12.235
print("{:<05d}".format(12))	12000
print("{:=8.3f}".format(-12.2346))	- 12.235
print("{:5}".format("cat"))	cat
print("{:>5}".format("cat"))	Cat
print("{:^5}".format("cat"))	cat
print("{:*^5}".format("cat"))	*cat*
print("{:.3}".format("caterpillar"))	Cat
print("{:5.3}".format("caterpillar"))	cat
print("{:^5.3}".format("caterpillar"))	cat
person = {'age': 23, 'name': 'Adam'}	Adam's age is: 23
print("{p[name]}'s age is: {p[age]}".format(p=person))	
person = {'age': 23, 'name': 'Adam'}	Adam's age is: 23
print("{name}'s age is: {age}".format(**person))	

string = "{:{fill}{align}{width}}"	*cat*
print(string.format('cat', fill='*', align='^', width=5))	
num = "{:{align}{width}.{precision}f}"	123.24
print(num.format(123.236, align='<', width=8, precision=2))	
print('%s %s' % ('one', 'two'))	one two
print('{} {}'.format('one', 'two'))	one two
print('%d %d' % (1, 2))	1 2
print('{} {}'.format(1, 2))	1 2
print('{1} {2} {0}'.format('one', 'two', 'three'))	two three one
print('%10s' % ('test',))	Test
print('{:>10}'.format('test'))	Test
print('%-10s' % ('test',))	test
print('{:10}'.format('test'))	test
print('{:_<10}'.format('test'))	test______
print('{:^10}'.format('test'))	test
print('{:^6}'.format('zip'))	zip
print('%.5s' % ('xylophone',))	Xylop
print('{:.5}'.format('xylophone'))	Xylop
print('%-10.5s' % ('xylophone',))	xylop
print('{:10.5}'.format('xylophone'))	xylop
print('%d' % (42,))	42
print('{:d}'.format(42))	42
print('%f' % (3.141592653589793,))	3.141593
print('{:f}'.format(3.141592653589793))	3.141593
print('%4d' % (42,))	42
print('{:4d}'.format(42))	42
print('%06.2f' % (3.141592653589793,))	003.14
print('{:06.2f}'.format(3.141592653589793))	003.14
print('%04d' % (42,))	0042
print('{:04d}'.format(42))	0042
print('%+d' % (42,))	+42

print('{:+d}'.format(42))	+42
print('% d' % ((- 23),))	-23
print('{: d}'.format((- 23)))	-23
print('% d' % (42,))	42
print('{: d}'.format(42))	42
print('{:=5d}'.format((- 23)))	- 23
print('{:=+5d}'.format(23))	+ 23
data = {'first': 'Hodor', 'last': 'Hodor!'}	Hodor Hodor!
print('%(first)s %(last)s' % data)	
print('{first} {last}'.format(**data))	Hodor Hodor!
print('{first} {last}'.format(first='Hodor', last='Hodor!'))	Hodor Hodor!
from datetime import datetime print('{:%Y-%m-%d %H:%M}'.format(datetime(2001, 2, 3, 4, 5)))	2001-02-03 04:05
print('{:{align}{width}}'.format('test', align='^', width='10'))	test
print('%.*s = %.*f' % (3, 'Gibberish', 3, 2.7182))	Gib = 2.718
print('{:.{prec}} = {:.{prec}f}'.format('Gibberish', 2.7182, prec=3))	Gib = 2.718
print('%*.*f' % (5, 2, 2.7182))	2.72
print('{:{width}.{prec}f}'.format(2.7182, width=5, prec=2))	2.72
print('{:{prec}} = {:{prec}}'.format('Gibberish', 2.7182, prec='.3'))	Gib = 2.72
print('{:{}{}{}.{}}'.format(2.7182818284, '>', '+', 10, 3))	+2.72
print('{:{}{sign}{}.{}}'.format(2.7182818284, '>', 10, 3, sign='+'))	+2.72

Functions

- A function is a block of organized, reusable code that is used to perform a single, related action.

- Functions provide better modularity for the application and a high degree of code reusing.

Defining a function

- Function blocks begin with the keyword **def** followed by the function name and 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.
    - Any input parameters or arguments should be placed within these parentheses.
    - 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.
    - By default, parameters have a positional behavior and have to be informed in the same order that they were defined.
        - **def functionname(parameters):**

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

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, execute it by calling it from another function or directly from the Python prompt.

Functions with Default Parameters

- Functions can use a default parameter value.
- If the function is called without parameter, it uses the default value.
    - **def Function_name(arg=value):**
    - **body**

Pass by reference vs value

- All parameters (arguments) in the Python language are passed by reference.
- If what a parameter refers to within a function is changed, the change also reflects back in the calling function.
- Example

Program	Output
# Function definition is here	
def changeme(mylist):	

"This changes a passed list into this function"	
mylist.append([1,2,3,4])	
print ("Values inside the function: ", mylist)	
Return	
# Now call changeme function	
mylist = [10,20,30]	
changeme(mylist)	Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
print ("Values outside the function: ", mylist)	Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

Functions Arguments

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

- Required arguments are the arguments passed to a function in correct positional order.
- The number of arguments in the function call should match exactly with the function definition

Keyword Arguments

- Keyword arguments are related to the function calls.
- When keyword arguments are used in a function call, the caller identifies the arguments by the parameter name.

Default Argument

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
- Example

Program	**Output**
def my_function(country = "Norway"):	
print("I am from " + country)	
my_function("Sweden")	I am from Sweden
my_function("India")	I am from India
my_function()	I am from Norway

my_function("Brazil")	I am from Brazil

Variable-length Arguments

- A function may process more arguments than specified specified arguments.
- These arguments are called variable-length arguments and are not named in the function definition,
- An asterisk (*) is placed before the variable name that will hold the values of all non-keyword variable arguments.
- This tuple remains empty if no additional arguments are specified during the function call.
    - **def functionname ([formal_args,] *var_args_tuple):**
    - **"function_docstring"**
    - **function_suite**
    - **return [expression]**
- Example

Program	Output
def sum1(*x):	
s = 0	
for i in x:	
s += i	
print("Sum of numbers is", s)	
sum1(1, 2, 3, 4)	Sum of numbers is 10
sum1(-1, 2, -3, 4)	Sum of numbers is 2
sum1()	Sum of numbers is 0
sum1(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)	Sum of numbers is 55

Scope of Variables

- The scope of a variable determines the portion of the program where a particular identifier can be accessed.
    - Global variables
    - Local variables
- The statement global VarName tells Python that VarName is a global variable.
- Python stops searching the local namespace for the variable.
- Example

Program	Output
total = 0	

# Function definition is here	
def sum (var1, var2):	
global total	
total = var1 + var2	
print ("Sum of 2 numbers printing in function", total)	Sum of 2 numbers printing in function 12
return total	
a = sum (5, 7)	
print ("Value of total variable: ", total)	Value of total variable: 12
print ("Value of total variable: ", a)	Value of total variable: 12

Recursive Functions

- Recursion is the process of defining something in terms of itself.
- A function can call other functions.
- It is even possible for the function to call itself.
- This type of construct is termed as recursive functions.
- Example

Program	Output
def calc_factorial(x):	
"""This is a recursive function	
to find the factorial of an integer"""	
if x == 1:	
return 1	
else:	
return (x * calc_factorial(x-1))	
num=None	
while(num!=-1):	Enter the number:5
num = int(input("Enter the number:"))	The factorial of 5 is 120
if(num <0):	Enter the number:4
print("Factorial is not possible")	The factorial of 4 is 24
Break	Enter the number:-1
print("The factorial of", num, "is", calc_factorial(num))	Factorial is not possible

Exception Handling

- Python has many built-in exceptions which forces the program to output an error when something in it goes wrong.
- When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled.
- If not handled, the program will crash.
  - If function A calls function B which in turn calls function C and an exception occurs in function C.
  - If it is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is spit out and the program comes to a sudden, unexpected halt.
- In Python, exceptions can be handled using a try statement.
- A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

  ```
  try:
     # do something
     pass
  except ValueError:
     # handle ValueError exception
     pass
  except (TypeError, ZeroDivisionError):
     # handle multiple exceptions
     # TypeError and ZeroDivisionError
     pass
  except:
     # handle all other exceptions
     pass
  ```
- In Python programming, exceptions are raised when corresponding errors occur at run time, but can be forcefully raised it using the keyword **raise**.
- It can be optionally passed in value to the exception to clarify why that exception was raised.
- The try statement in Python can have an optional finally clause.
- This clause is executed no matter what, and is generally used to release external resources.
- A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.

Program	Output

# import module sys to get the type of exception	The entry is a
import sys	Oops! <class 'ValueError'> occured.
randomList = ['a', 0, 2]	Next entry.
for entry in randomList:	
try:	The entry is 0
print("The entry is", entry)	Oops! <class 'ZeroDivisionError'> occured.
r = 1/int(entry)	Next entry.
Break	
except:	The entry is 2
print("Oops!",sys.exc_info()[0],"occured.")	The reciprocal of 2 is 0.5
print("Next entry.")	
print()	
print("The reciprocal of",entry,"is",r)	
try:	
a = int(input("Enter a positive integer: "))	Enter a positive integer: -9
if a <= 0:	That is not a positive number!
raise ValueError("That is not a positive number!")	
except ValueError as ve:	Enter a positive integer: 9
print(ve)	
try:	
raise KeyboardInterrupt	
finally:	Goodbye, world!
print('Goodbye, world!')	