Basic Python Programming Concepts to solve Exercise Problems (P_VTP1)

11/7/2020

Contents

- Value and Built-in Type
  - P_VTP1: 10
- Type Conversion
  - P_VTP1: 3, 4
- Difference between end and sep
  - P_VTP1: 11
- Operators
  - P_VTP1: 5, 6, 7, 8
- String
  - P_VTP1: 2, 23, 24, 25, 26, 27, 28, 40, 41, 42
- Lists
  - P_VTP1: 36, 37, 38, 39

- Tuples
  - P_VTP1: 9, 29, 30, 31, 32, 33, 34, 35
- Dictionary
  - P_VTP1: 43, 44, 45, 46, 47
- Selection Construct
  - P_VTP1: 12, 13, 14
- Iteration Constructs
  - P_VTP1: 15, 16, 17, 18, 19
- Break, Continue, Pass
  - P_VTP1: 20, 21, 22
- Correspondence Between Each Topic and Related VTPs
- Conclusion

Correspondence Index

Basic Grammar Concept Topic	Related Slide Number
String	28
Lists	33
Tuples	39
Dictionary	45
Selection Construct	49
Iteration Construct	54
Break Statement	58
Continue Statement	59
Pass Statement	60
Correspondence Between Each Topic and Related VTPs	61
Conclusion	62

What is Python ?

- Python is a **high-level programming** language, with applications in numerous areas, including web programming, scripting, scientific computing, and artificial intelligence.

- It is **very popular** and used by organizations such as Google, NASA, U tube, I-robot, Intel, Cisco, etc.

- Python is easy to use, powerful, and versatile, making it a **great choice** for beginners and experts.

- Therefore, we need to take **a lot of study** for future learning.

Different Syntax usage of Python from Java

```java
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```


```python
print("Hello world!")
```

Few Important Things to Remember

- To represent a statement in Python, **newline** (enter) is used.
- **Use of semicolon** at the end of the statement is **optional** (unlike languages like C/C++).
- In fact, it's recommended to **omit semicolon** at the end of the statement in Python.
- Instead of curly braces { }, **indentations**(number of tab) are used to represent a block.

Commenting in Python

- Python **single line comment** preceded by a hash symbol (#)

Example:

```
#This is a comment
print("Hello, World!")
```

- Three consecutive single quotation marks ‘’’ are used to give **multiple comments** (or) paragraph comments.

Example:

```
""" This is a comment
written in
more than just one line """
print("Hello, World!")
```

Variables and its Names

- A variable allows to **store a value** by assigning it to a name, which can be used to refer to the value **later** in the program.

- **Rules** for Python variables:

  - **Must start** with a letter or the underscore character

  - **Cannot start** with a number

  - **Should be** meaningful and short, case-sensitive(age, Age are not the same)

Data Types

- Different types in python are:

(1) Numbers (int, float, etc >> 3, 4.5, etc)

(2) Lists >> [1, 2, 3, 4], ["Hello", "world!"], [1, 2, "Hello"]

(3) Tuples >> (1, 2), ("hi", "hello", "bye"), (2, "Lucy", 45)

(4) Strings >> "Hello world!", ‘K3WL’

(5) Sets >> {"apple", "banana", "cherry"}

(6) Dictionary >> {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}

Value and Built-in Type

- To know type of any value in Python use **in-build method** called **type(value)**.

Example:

type(‘123’) return build-in type as <class 'int'>
type(‘123.33’) return build-in type as <class 'float'>
type(‘hello’) return build-in type as <class 'str'>
type(‘True’) return build-in type as <class 'bool'>
type(‘3+4j’) return build-in type as <class 'complex'>

Type Conversion

- It is **possible to change** one type of value/ variable to another type. It is known as type conversion or type casting.

- Therefore, casting in python is done using **constructor functions**.

Example:

```
a= 12.34
b= int(a)
print (b ) >> The result is 12.
```

- Python automatically converts one data type to another data type. This process doesn't need any user involvement. This is called **Implicit type conversion**.

Example

```
num_int = 13
num_flo = 1.1
num_new = num_int + num_flo
print("datatype of num_int:",type(num_int)) >> <class 'int'>
print("datatype of num_flo:",type(num_flo)) >> <class 'float'>
print("Value of num_new:",num_new) >> 14.1
print("datatype of num_new:",type(num_new)) >> <class 'float'>
```

Difference between end and sep

- end and sep are optional parameters of Python.
- The end parameter basically prints after all the output objects present in one output objects present in one output statement have been returned.
- The sep parameter differentiates between the objects.

EXAMPLE:

```
a=2
b='abc'
print(a,b,sep=',')
print(a,b,end=',')
```

OUTPUT:

```
2,abc
2 abc,
```

Operators

- Python divides the operators in the following groups:

  - Arithmetic Operators
  - Assignment Operators
  - Comparison Operators
  - Logical Operators
  - Relational Operators
  - Bitwise Operators
  - Identity Operators
  - Membership Operators

Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common **mathematical operations**.

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiatio	x ** y
//	Floor division	x // y

- **Example**

```
x = 33
y = 2
# Output for addition of x and y
print('x + y = {}'.format(x+y)) >> 35
# Output for subtration of x and y
print('x - y = {}'.format(x-y)) >> 31
# Output for multiplication of x and y
print('x * y = {}'.format(x*y)) >> 66
# Output for division of x and y
print('x / y = {}'.format(x/y)) >> 16.5
# Output for modulus of x and y
print('x // y = {}'.format(x//y)) >> 16
# Output for exponent of x and y
print('x ** y = {}'.format(x**y)) >> 1089
```

Assignment Operators

- Assignment operators are used **to assign values** to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

- Comparison operators are used **to compare two values**:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

- **Example**

```
# Output: x > y is True or False
print('x > y is',x>y) >> False
# Output: x == y is True or False
 print('x == y is',x==y) >> False
# Output: x != y is True or False
print('x != y is',x!=y) >> True
# Output: x >= y is True or False
print('x >= y is',x>=y) >> False
# Output: x <= y is True or False
print('x <= y is',x<=y) >> True
```

Logical Operators

- Logical operators are used **to combine conditional statements**.

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

- **Example**

  x = False

  y = True

  print('x and y is',x and y) >> False

  print('x or y is',x or y) >> True

  print('not x is',not x) >> True

Identity Operators

- Identity operators are used **to compare the objects.**

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

- **Example**

```
x1 = 3
y1 = 3
x2 = 'Welcome'
y2 = 'Welcome'
x3 = [1,2,3,4]
y3 = [1,2,3]
print("x1 is not y1", x1 is not y1) >> False
print("x2 is y2", x2 is y2) >> True
print("x3 is y3", x3 is y3) >> False
```

Bitwise Operators

- Bitwise operators are used **to compare (binary) numbers**.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits

String

- String is a sequence of characters, like "Python is cool"
- Each character has an index

P	y	t	h	o	n		i	s		c	o	o	l
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Accessing a character: `string[index]`

```
x = "Python is cool"
print(x[10]) // Output is c.
```

- Accessing a substring via slicing: `string[start:finish]`

```
print(x[2:6]) // Output is thon.
```

String Operations

P	y	t	h	o	n		i	s		c	o	o	l
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
>>> x = "Python is cool"
>>> "cool" in x        # membership    >> True
>>> len(x)             # length of string x   >> 14
>>> x + "?"            # concatenation  Python is cool?
>>> x.upper()          # to upper case
>>> x.replace("c", "k") # replace characters in a string
```

- subscripting of strings

'Hello'[2] → 'l'

slice: 'Hello'[1:2] → 'el‘

'Hello'[:2] → ‘He‘

'Hello'[2:] → ‘llo‘

'Hello'[2:-1] → ‘llo'

word[-1] → last character ‘o’

len(word) → 4

‘Hello’* 3 → HelloHelloHello (String Replication)

Relational Operator on String

- The ASCII value of a is 97, b is 98 and so on.
- The ASCII value of A is 65, B is 66 and so on.

Example

str1 = 'A'

str2 = 'B'

str3 = 'a'

str4 = 'b'

print (“str1>str3”, str1>str3) ,The output is False.

print (“str2> str1”, str2>str1), The output is True.

String Format

- Strings and numbers can be combined by using the **format() method.**

- The **format() method** takes the passed arguments, formats them, and places them in the string where the placeholders {} are.

Example:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Output: My name is John, and I am 36

Lists

- A list is created by placing all the items (elements) **inside a square bracket []**, separated by commas.

- It can have any number of items, and they may be of **different types** (integer, float, string, etc.)

- Lists are mutable, meaning, their elements **can be changed**.

Creating List

Example

#empty list >> my_list = []

#list of integers >> my_list = [1,2,3]

#list with mixed data types>> my_list = [1, “Hello”, 3.4]

#nested list >> my_list = [“Welcome”, [8,4,6]]

Accessing Items from a list

- Use the index operator **[]**

Example

list = [‘p’,’r’,’o’,’b’,’e’]

positive indexing

Print (list[**2**]) >> o

negative indexing and negative index -1 refers to the last item

Print (list[**-2**]) >> b

Slicing operation on list

Print(list**[1:3]**) >> r,o

nested list

list = [“welcome”, [8,4,6]]

Print(list**[1][0]**) >> 8

Change or Add Elements to a list

#**Change Elements**

```
marks=[90,60,80,66,76,45,60]
marks[1]=100
marks[3:6] = [11, 22, 33]
print(marks) >>>[90, 100, 80, 11, 22, 33, 60]
```

#**Add Elements**

-add one item to a list using **append()** method

-add several items using **extend()**

-insert one item at a desired location by **insert()** method

```
marks.append(50)
print(marks) >>>[90, 100, 80, 50]
marks.extend([60,80,70])
print(marks) >>>[90, 100, 80, 50, 60, 80, 70 ]
marks.insert(3,40)
print(marks) >>>[90, 100, 80, 40, 50, 60, 80, 70]
```

Delete or Remove Elements from a List

- **del keyword** to delete one or more items from a list.
  marks = [90, 100, 80, 40, 50, 60, 80, 70]
  del marks[6]
  print(marks) >>> [90, 100, 80, 40, 50, 60, 70]
  del marks[2:4] >>> [90,100,50,60,70]
- **clear() method** to empty a list.
  marks.clear()
  print(marks) >>> []
- **remove() method** to remove the given item
  marks.remove(50)
  print(marks) >>> [90, 100, 60, 70]

- **pop() method** to remove an item at the given index.

```
marks=[100,20,30]
marks.pop() >>> 30
print(marks) >>> [100, 20]
marks.pop(0) >>> 100
print(marks) >>>[20]
```

Tuples

- Tuples are very similar to lists, except that they are **immutable** (they cannot be changed).

- They are created **using parentheses**, rather than square brackets.

Creating a Tuple

```
# empty tuple
my_tuple = ()
print(my_tuple)

# tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)

# tuple can be created without parentheses
# also called tuple packing
my_tuple = 3, 4.6, "dog"
print(my_tuple)
```

```
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
(3, 4.6, 'dog')
```

Accessing Elements in a Tuple

- Can access the values with **their index.**
- Nested tuple are accessed using **nested indexing**.
- A range of items can be accessed by using **slicing operator**.

```
marks = (23,45,32,44,30,80,70)
print (marks[0]) >>> 23
print (marks[-2]) >>> 80
print (marks[1:5]) >>> 45,32,44,30

# nested tuple
n_tuple = ("Skin", [8,4,6], (1,2,3))
print (n_tuple[0]) >>> Skin
print(n_tuple[1]) >>> [8,4,6]
print (n_tuple[0][0]) >>> S
print(n_tuple[1][0]) >>> 8
```

- **Iteration on Tuple**

Example

```
# tuple of names
my_tuple = (“John", “Smitt", “Roy San", “Carlk")
 # iterating over tuple elements
 for name in my_tuple:
        print(name)
```

Then, the output is John, Smitt, Roy San, Carlk.

Changing a Tuple

- If the element is itself a **mutable data type like list**, its nested items can be changed.

Example

```
n_tuple = ("Skin", [8,4,6], (1,2,3))
n_tuple[1][1] = 23
print (n_tuple) >>> ('Skin', [8,23,6], (1,2,3))
```

- **+ operator** can be used to combine two tuples.
- *** operator** can be used to repeat the elements in the tuples for a given number of times.

Example

```
# Concatenation
   print ((1,2,3)+(4,5,6)) >>> (1,2,3,4,5,6)
# Repeat
   print (("Repeat")*2) >>> ('Repeat', 'Repeat')
```

Tuple Membership Test

- **in keyword** is used to test if an item exists in a tuple or not.

Example

```
my_tuple = ('a','p','p','l','e')
print('a' in my_tuple) >>> True
print('b' in my_tuple) >>> False
print('g' not in my_tuple) >>> True
```

Iterating Through a Tuple

```
names = ("John", "Kate", "Shan")
for name in names:
    print('Hello', name)
```

Output >>> Hello John
Hello Kate
Hello Shan

Dictionary

- A dictionary is mutable and is **another container type** that can store any number of Python objects.

- It consists of **pairs** (called items) of **keys** (unique) and their corresponding **values**.

- The values can be of **any type**, but the keys must be of **an immutable data type** such as strings, numbers, or tuples.

- The general **syntax of a dictionary** is as follows:

```
dict = {'A': '2341', 'B': '9102', 'C': '3258'}
```

Accessing Values in Dictionary

- To access dictionary elements, the familiar **square brackets along with the key** can be used **to obtain its value**.

Example

```
dict = {‘Name’: ‘Zara’, ‘Age’: 7, ‘Class’: ‘First’};
print ( dict[‘Name’]) >>> Zara
# loop through dictionary
for e in mydict:
    print("Key:",e,"Value:",dict[e])
```

Then, the output is Key: Name Value: Zara
Key: Age Value: 7
Key: Class Value: First

Updating Dictionary

- A dictionary can be updated by **adding** a new entry or item (i.e., a key-value pair), **modifying** an existing entry or deleting an existing entry.

Example

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print("dict['Age']:", dict['Age']); >>> 8
print("dict[School']:", dict['School']); >>> DPS School
```

Delete Dictionary Elements

- **Individual** dictionary elements can either be removed or the **entire contents** of a dictionary can be cleared.

Example

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
del dict['Name']; # remove entry with key 'Name'
dict.clear(); # remove all entries in dict
del dict; # delete entire dictionary
```

Selection Construct

- It is also known as **conditional construct**. This structure helps the programmer to take **appropriate decision**.

- There are **five kinds** of selection constructs

1. Simple – if
2. if – else
3. elif
4. Nested – if
5. Multiple Selection

1. Simple-if

- The **general form** of simple – if statement is:

```
if x > 0:
        Statement 1
Statement 2
```

- Here, if the result of the test condition is TRUE then the Statement 1 is executed. Otherwise, Statement 2 is executed

2. if-else Statement

- This structure helps to decide whether **a set** of statements should be executed or **another set** of statements should be executed. This statement is also called as **two-way branch.**

- The general form of if – else statement is:

```
if (Test Condition A):
        Statement B
else:
        Statement C
```

- Here, If the test-condition is TRUE, statement-B is executed. Otherwise, Statement C is execute

3. elif Statement

- This structure helps the programmer to decide the **execution of a statement from multiple statements** based on a condition.
- There will be **more than one condition to test**. This statement is also called as multiple-way branch.
- The **general form** of if – else – if statement is:

```
if (Test Condition 1):
    Statement 1;
elif (Test Condition 2):
    Statement 2
        ……
elif( test Condition N):
    Statement N
```

4. **Nested if statement**

- The statement **within the if statement is another if statement** is called Nested– if statement.
- The **general form** of Nested – if statement is:

- **if** (Test Condition 1):

```
    if (Test Condition 2):
            Statement 1
     else:
            Statement 2;
else:
     if (Test Condition 3):
            Statement 3;
     else:
            Statement 4;
```

Iteration Constructs

- Python provides two types of looping constructs:

(1) While statement
(2) For statement

- **While**: The set of statements are **executed repeatedly until** the condition is true. When it becomes false, control is transferred **out of the structure**.

- The **general form** of while structure is:

While (Test Condition):
    Statement 1
    Statement 2
    ……..
    Statement N
[else: # optional part of while
STATEMENTs BLOCK 2]

- **Nested loops**
  Block of statement belonging to **while can have another while** statement, i.e. a while can contain another while.
  Example

```
i=1
while i<=3:
        j=1
        while j<=i:
                print j, # inner while loop
                j=j+1
        print i=i+1
```

  The output is printed as:

```
>>>1
1 2
1 2 3
```

For Loop

- This structure is usually used when we **know in advance exactly how many times** asset of statements is to be repeatedly executed repeatedly.
- It can be used as **increment** looping or **decrement** looping structure.
- The **general form** of for structure is as follows:

- **for i in range (initial value, limit, step):**
  STATEMENT BLOCK 1
  [else: # optional block
  STATEMENT BLOCK 2]
  Example
  # loop to print value 1 to 10
  for i in range (1, 11, 1):
  print (i) # the output is 1,2,3,4,5,6,7,8,9,10

Break Statement

- Break can be used **to unconditionally jump out** of the loop.
- It **terminates the execution** of the loop.

- **Example**

```
for letter in ‘’Python’’:
        if letter = = ‘’h’’:
            break
        print letter
```

The output will be printed like:

P y t

Continue Statement

- This statement is used to skip the rest of the statements of the current loop block and to move to next iteration, of the loop.

- **Example:**

```
for letter in “Python”:
    if letter == “h”:
        continue
    print letter
```

Will result into

P y t o n

Pass Statement

- pass does nothing

- **Example :**

```
i = 3
if i==3:
        pass;
        print("\nWe are inside pass block\n")
```

The output will be printed like We are inside pass block.

Correspondence Between Each Topic and Related VTPs

Basic Grammar Concepts	Related Problem Number
Data Types	1
Value and Built-in Type	10
Type Conversion	3,4
Difference between end and sep	11
Operators	5,6,7,8
String	2,23,24,25,26,27,28,40,41,42
Lists	36,37,38,39
Tuples	9,29,30,31,32,33,34,35
Dictionary	43,44,45,46,47
Selection Construct	12,13,14
Iteration Construct	15,16,17,18,19
Break, Continue, Pass	20,21,22

Index Page

Conclusion

- This presentation includes basic concepts for python Programming.
- I hope that it will help for your study.