Introduction to the Course

Lecture 0

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- **Course Number and Name:**

  CSE 3142, **`Programming in Python`**

- **Credits and Course Format:**

  Grading Pattern = 5

  Credits = 3

  Course format: 6 hours/week (2 labs/week, 2hrs/lab, 1 Problem solving class/week, 2hrs/class)

- **Target Students:**

  Programme: B.Tech. (5^{th} Semester)

  Branch: CSE

- **Text Book(s):**

  (1) Python Programming, A modular approach by Sheetal Taneja and Naveen Kumar, Pearson India

Course Information

- **Specific Course Information:**
  (a) **Course Description:** Python Programming: An Introduction, Functions, Control Structures, Debugging, Scope, Strings, Mutable and Immutable Objects, Recursion, Files and Exceptions, Classes I, Classes II, List Manipulation, Data Structures I: Stack and Queues, Data Structures II: Linked Lists, Data Structure III: Binary Search Trees
  (b) **Prerequisites:** Introduction to Computer Programming (CSE 1001), Data Structure and Algorithms (CSE 2001)

Course Outcomes(COs)

- **Course Outcomes (COs) :**

  By the end of course through lectures, readings, homeworks, laboratory, assignments and exams students will be able to:

CO 1. Understand the basic programming syntax, semantics and building blocks of python.

CO 2. Develop python applications using the programming constructs like control structures, functions and strings.

CO 3. Analyze, debug and test the programs and correctly predict their output.

CO 4. Illustrate the process of structuring the data using lists, sets, tuples and dictionaries.

CO 5. Solve the real life problems using object oriented concepts, modular approaches, files and exception handling.

CO 6. Design application using sorting, searching and the concept of stack, queues, linked lists and trees.

Evaluation scheme

- **Evaluation scheme (under Grading Pattern-5) out of** 100**%:**

Attendance:	05%
Weekly Assignments/Quizzes :	20%
Mid-Term :	15%
End-Term Examination:	60%

Program Outcomes

- There are twelve program outcomes (1-12) and two program specific outcomes for the Computer science & Engineering B. Tech program:

  Program Outcomes (POs)

  1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
  2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

Program Outcomes Contd..

3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

Program Outcomes Contd..

5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

Program Outcomes Contd..

7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

Program Outcomes Contd..

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

Program Outcomes Contd..

12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

PSO1: The ability to understand, analyze and develop computer programs in the areas related to business intelligence, web design and networking for efficient design of computer-based systems of varying complexities.

PSO2: The ability to apply standard practices and strategies in software development using open-ended programming environments to deliver a quality product for business success.

Summary

Thank You

Any Questions?

Python Programming-An Introduction

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- Python is a general purpose, high level and interpreted programming language.
- Python is an interactive programming language.
- Simple syntax, Easy to read and write
- Python was developed by **Guido Van Rossum** in 1991 at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Introduction (Cont.)

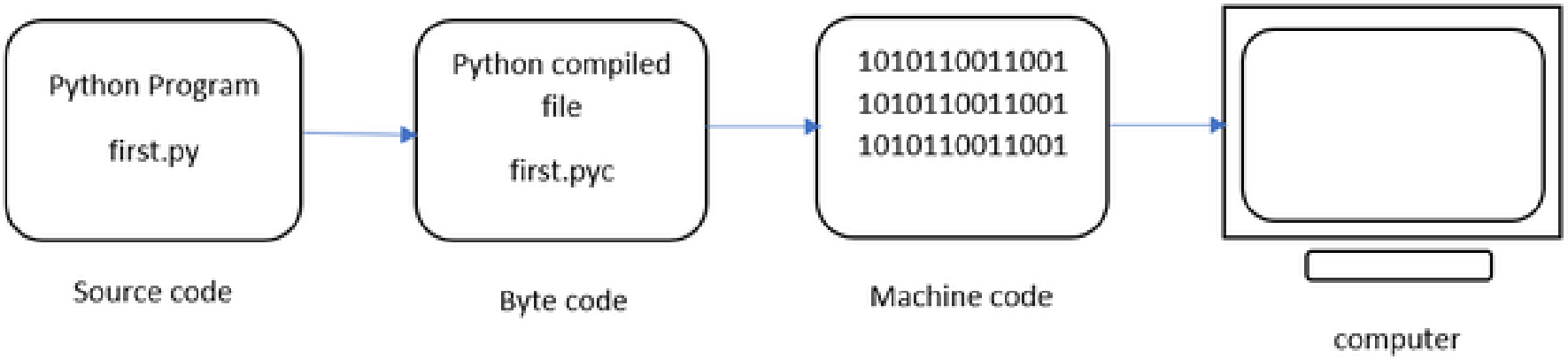


Figure 1: Steps of execution of a python program

Execution of a Python Program

- Normally, when we compile a python program, we cannot see .pyc file produced by python compiler and the machine code generated by Python Virtual Machine (PVM). This is done internally in the memory and the output is finally visible. For example, if our python program name is first.py, we can use python compiler to compile it as:
  python first.py
- In the preceding statement, *python* is the command for calling the python compiler. The compiler should convert the first.py file into its byte code equvalent file first.pyc. Instead of doing this, the compiler directly displays the output or result. To separately create .pyc file from the source code, we can use the following command:
  python -m py_compile first.py
- In order to interpret the .pyc file using PVM, the python compiler can be called using the following command:
  python first.cpython-38.pyc

IDLE - An Interpreter for Python

- IDLE stands for Integrated Development and Learning Environment.
- Python IDLE comprises **Python Shell** (An Interactive Interpreter) and **Python Editor** (Allows us to work in Script Mode).
- While using Python shell, we just need to type Python code at the ⟩⟩⟩ prompt and press Enter key. For Example:
  ⟩⟩⟩ print('Hello World')
  Hello World
- Python shell may also be used as a calculator. For Example:
  ⟩⟩⟩ 18+5
  23
  ⟩⟩⟩ 27//5
  5
  ⟩⟩⟩ 27.0//5
  5.0
  ⟩⟩⟩ 27%5
  2

IDLE - An Interpreter for Python

- We evaluate the foregoing expressions in Python shell. For Example:

```
>>> 3**2
9
>>> 6/3/2          #(6/3)/2
1.0
>>> 2**3**2        #2**(3**2)
512
```

- Left associative operators: +, -, *, /, //, %
- Right associative operators: **
- Precedence of Arithmetic operators are:

() (parentheses)	decreasing order
** (exponentiation)	
- (negation)	
/ (division) // (integer division) * (multiplication) % (modulus)	
+ (addition) - (subtraction)	

IDLE - An Interpreter for Python

- While the parentheses have the highest precedence, addition and subtraction are at the lowest level.
- Python complains when it encounters a wrongly formed expression. For Example
  ⟩⟩⟩ 7+3(4+5)
  Traceback (most recent call last):
  File "⟨*pyshell*#17⟩", line 1, in ⟨*module*⟩
  7+3(4+5)
  TyeError: 'int' object is not callable
- Similarly, Division by zero is a meaningless operation. For Example
  ⟩⟩⟩ 7/0
  Traceback (most recent call last):
  File "⟨*pyshell*#18⟩", line 1, in ⟨*module*⟩
  7/0
  ZeroDivisionError: division by zero

Python Strings

- A string is a sequence of characters.
- We can enclose a a sequence of characters between single, double, or triple quotes to specify.
- A string enclosed in single quotes may include double quotes marks and vice versa.
- A string enclosed in triple quotes (also known as docstring, i.e. documentation string) may include both single and double quote marks and may extend over several lines. For Example
  ⟩⟩⟩ 'Hello World'
  'Hello World'
  ⟩⟩⟩ print('Hello World')
  Hello World
  ⟩⟩⟩ """Hello
  World"""
  "Hello
  World"

Python Strings

- Escape sequence \ n marks a new line.
- Use of + as the concatenation operator.
- The operator * (multiplication) is used to repeat a string a specified number of times.
- For Example:
  ⟩⟩⟩ 'hello' + '!!'
  'hello !!'
  ⟩⟩⟩ 'how' + 'are' + 'you?'
  'how are you?'
  ⟩⟩⟩ 'hello' * 5
  'hellohellohellohellohello'

Relational Operators

- Relational Operators are used for comparing two expressions and yield True or False.
- Arithmetic operators have higher precedence than the relational operators.

==	(equal to)
<	(less than)
>	(greater than)
< =	(less than or equal to)
> =	(greater than or equal to)
! =	(not equal to)

- The relational operators are:
- A relational operator applied on expressions as:
  expression ⟨*comparisonoperator*⟩ expression
- For Example:
  ⟩⟩⟩ 23 != 23
  False
- ASCII vales of characters are used for string comparison.
- Python 3 does not allow string values to be compared with numeric values.

Logical Operators

- Logical operators not, and, and or are applied to logical operands True and False, also called Boolean values, and yield either rue or False.
- The operator not is unary, whereas other two are binary operator, those are as described below:

not	
True	False
False	True

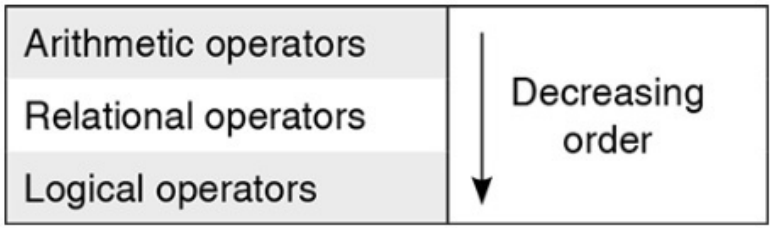
and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

Logical Operators

- The precedence of logical operators are:

- The precedence of operators are:



- For Example: The expression (10 $\langle$ 5) and ((5/0) $\langle$ 10) yields False, the expression (10 $\rangle$ 5) and ((5/0) $\langle$ 10) yields an error.

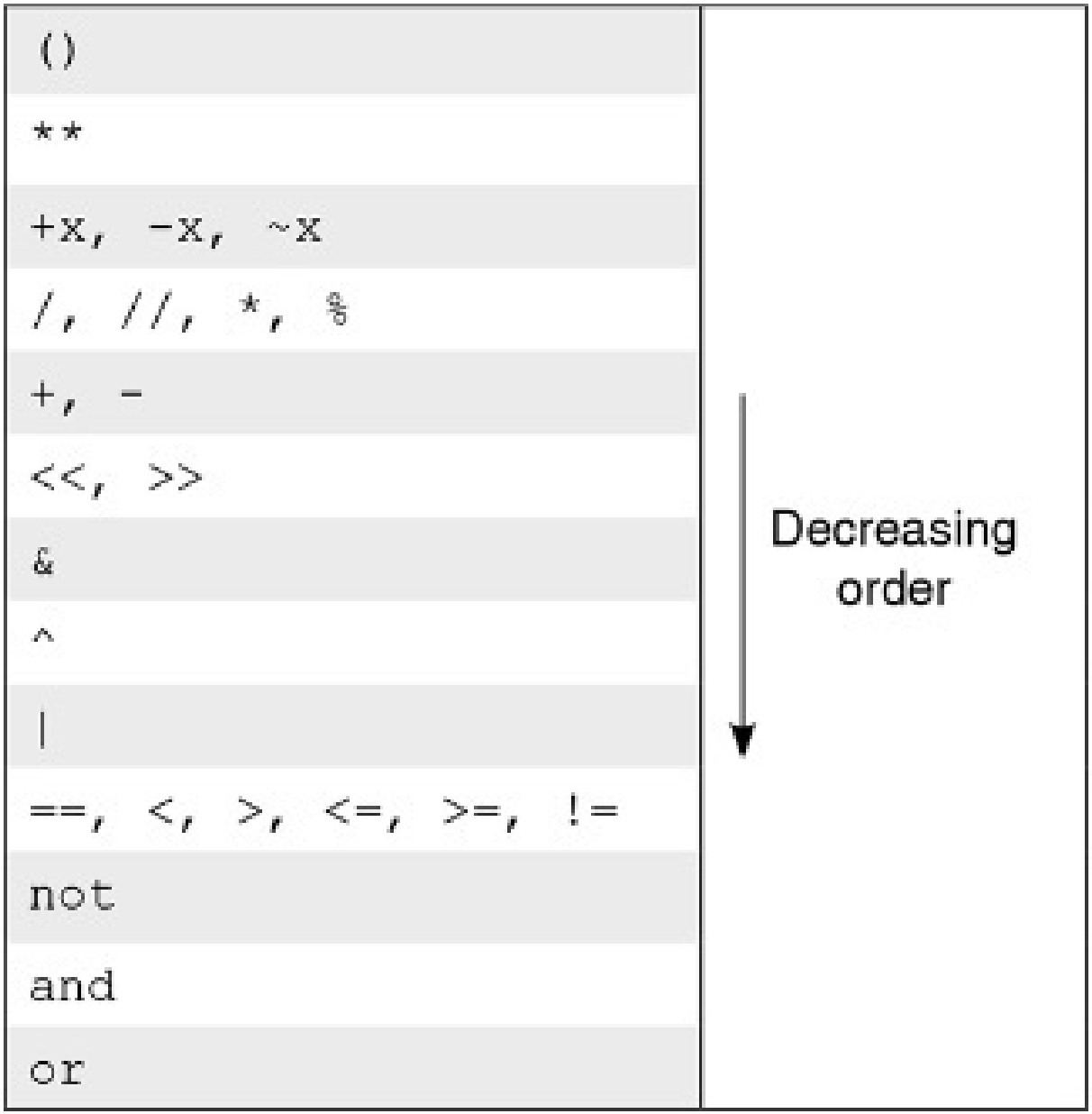
Bitwise Operators

- These operators operate on integers interpreted as strings of binary digits 0 and 1, also called bits, as:

Bitwise operator	Description	Example
Bitwise AND `x & y`	A bit in `x & y` is 1 if the corresponding bit in each of `x` and `y` is 1, and 0 otherwise.	`10 & 6` yields 2: 10: 00001010 6: 00000110 --- 2: 00000010
Bitwise OR `x | y`	A bit in `x | y` is 1 if at least one of the corresponding bits in `x` and `y` is 1, and 0 otherwise.	`10 | 6` yields 14: 10: 00001010 6: 00000110 --- 14: 00001110
Bitwise Complement `~ x`	A bit in `~x` is 1 if and only if the corresponding bit in `x` is 0. The result obtained from this operation is `-x-1`.	`~ 11` yields -12: 11: 00001011 --- -11: 11110100 Note that 11110100 is the two's complement of 12: 00001100.
Bitwise Exclusive OR `x ^ y`	A bit in `x ^ y` is 1 if exactly one of the corresponding bits in `x` and `y` is 1, and 0 otherwise.	`10 ^ 6` yields 12: 10: 00001010 6: 00000110 --- 12: 00001100
Left Shift `x << y`	Bits in the binary representation of `x` are shifted left by `y` places. Rightmost `y` bits of the result are filled with zeros. The result obtained from this operation is `x*2^y`.	`5 << 2` yields 20: 5: 00000101 --- 20: 00010100
Right Shift `x >> y`	Bits in the binary representation of `x` are shifted right by `y` places. Leftmost `y` bits of the result are filled with sign bit. The result obtained from this operation is `x/2^y`.	`5 >> 2` will yield 1 as the result as shown below: 5: 00000101 --- 1: 00000001

Bitwise Operators

- The precedence of operators are as:



Variables and Assignment Statements

- Variables provide a means to name values so that they can be used and manipulated later on. For Example:
  ⟩⟩⟩ english = 57
  When this statement is executed, Python associates the variable english with value 57.
- In Python style (often called Pythonic style or Pythonic way), variables are called names, and an assignment is called an association or a building.
  ⟩⟩⟩ english
  57
  ⟩⟩⟩ maths = 64
  ⟩⟩⟩ total = english + maths
  121

Variables and Assignment Statements (Contd..)

- Syntax for assignment statement:
  variable = expression
- Rules for naming variables:
  1. A variable must begin with a letter or _ (Underscore character).
  2. A variable may contain any number of letters, digits, or underscore characters. No other character apart from these is allowed.
- Always use meaningful variables names.
- Follow a consistent style in choosing variables.
- Examples of not valid variables as:

total_no.	#use of dot (.)
1st_number	#begins with a digit
AmountIn$	#use of dollar symbol ($)
Total Amount	#Presence of blank between two words

Variables and Assignment Statements (Contd..)

- Python is case-sensitive. For example, age and Age are not same, different.
- More than one variable may refer to the same object.
  ⟩⟩⟩ a = 5
  ⟩⟩⟩ b = a
  ⟩⟩⟩ b
  ⟩⟩⟩ a = 7
  ⟩⟩⟩ a
- The shorthand notation works for all binary mathematical operators. a = a ⟨*operator*⟩ b is equivalent to a ⟨*operator*⟩ = b. For example: a = a + b is equivalent to a + = b.
- Multiple assignments in a single statement as:
  ⟩⟩⟩ a, b, c = 'how', 'are', 'you?'
- Assigning same value to multiple variables in a single statement as:
  ⟩⟩⟩ a = b = 0

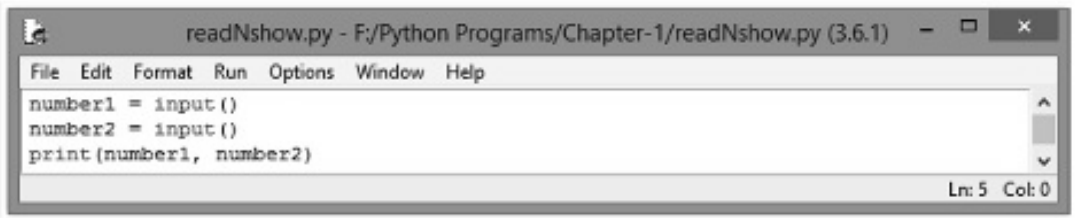
Keywords

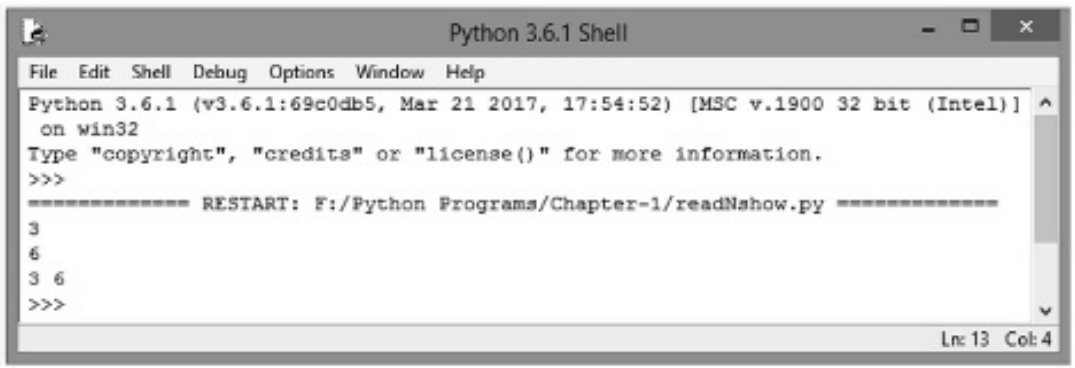
- Keywords are the reserved words that are already defined by the Python for specific uses.
- Keywords cannot be used for any other purposes.
- Keywords cannot be used for naming objects.
- List of Python keywords are as:

False	class	finally	is	return	None
continue	for	lambda	try	True	def
from	nonlocal	while	and	del	global
not	with	as	if	or	yield
assert	else	import	pass	break	except
in	raise	elif			

Script Mode

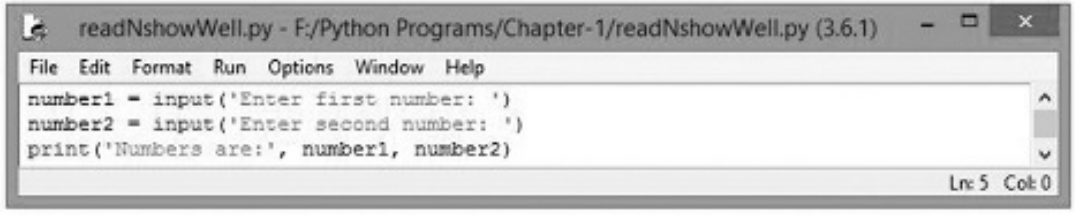
- When we exit an IDLE session, and start another IDLE session, we must redo all computations.This is not convenient mode of operation for most of the computational tasks.
- Python provides another way of working called script mode.
- In script mode, instructions are written in a file.
- A script should have extension .py or .pyw.
- For Example:

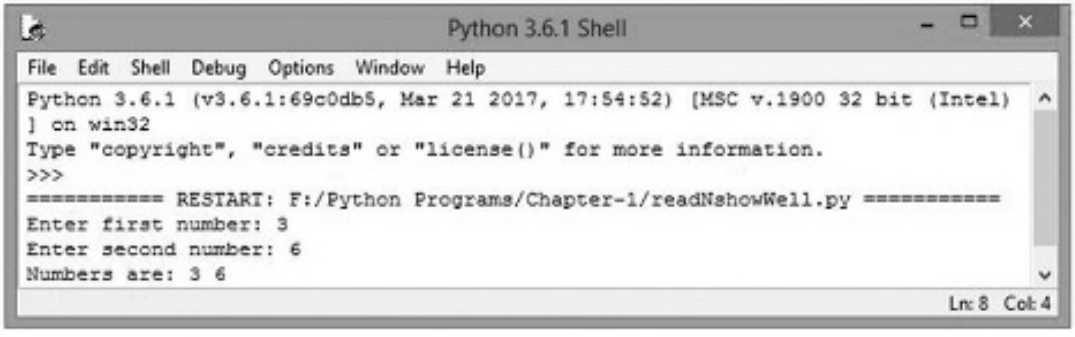




Script Mode (Contd..)

- We can modify the previous example as we may not remember that the program requires two numbers as the input.
- A good programming practice is to display to the user what data to be entered.
- For Example:





Summary

- Python interpreter executes one expression (command) at a time.
- A string is a sequence of characters. To specify a string, we may use single, double, or triple quotes.
- While evaluating a boolean expression involving *and* operator, the second sub-expression is evaluated only if the first sub-expression yields True.
- While evaluating an expression involving *or* operator, the second sub-expression is evaluated only if the first sub-expression yields False.
- A variable is a name that refers to a value. We may also say that a variable associates a name with data object such as number, character, string or Boolean.
- A variable name must begin with a letter or _.
- Python is case-sensitive.
- Python programming can be done in an interactive and script mode.

References

Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?

Functions

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- Simple statements can be put together in the form of functions to do useful tasks.
- To solve a problem, divide it into simpler sub-problems.
- The solutions of the sub-programs are then integrated to form the final program.
- This approach to problem solving is called step-wise refinement method or **modular approach**.
- Functions are generally of two types. Such as:
  1. Built-in functions
  2. User-defined functions

Built-in Functions

- Built-in Functions are predefined functions that are already available in Python.
- The function ***input*** enables us to accept an input string from the user without evaluating its value.
- The function *input* continues to read input text from the user until it encounters a newline. For Example:
  ⟩⟩⟩ name = input('Enter a Name:')
  Enter a Name: Alok
  ⟩⟩⟩ name
  'Alok'
  Here, the string 'Enter a Name:' specified within the parentheses is called an argument.
- The function eval is used to evaluate the value of a string. For Example:
  ⟩⟩⟩ eval('15')
  15
  ⟩⟩⟩ eval('15 + 10')
  25

Built-in Functions (Cont.)

- The value returned by a function may be used as an argument for another function in a nested manner. This is called **composition**.
  For Example:
  ⟩⟩⟩ n1 = eval(input('Enter a Number:'))
  Enter a Name: 234
  ⟩⟩⟩ n1
  234
- We can print multiple values in asingle call to print function, where expressions are separated by comma. For Example:
  ⟩⟩⟩ print(2, 567, 234)
  2 567 234
  ⟩⟩⟩ name = 'Raman'
  ⟩⟩⟩ print('hello',name, '2+2 =', 2+2)
  hello Raman 2+2 = 4

Built-in Functions (Cont.)

- We can use python escape sequences such as \n (newline), \t (tab), \a (bell), \b (backspace), \f (form feed), and \r (carriage return). For Example:
  ⟩⟩⟩ print('hello',name, '\n 2+2 =', 2+2)
  hello Raman
  2+2 = 4
- Python function type tells us the type of a value. For Example:
  ⟩⟩⟩ print(type(12), type(12.5), type('hello'), type(int))
  ⟨*class' int'*⟩, ⟨*class' float'*⟩, ⟨*class' str*⟩, ⟨*class' type'*⟩
- The round function rounds a number up to specific number of decimal places. For Example:
  ⟩⟩⟩ print(round(89.625,2), round(89.625), round(89.625,0)
  89.62 90 90.0

Built-in Functions (Cont.)

- The input function considers all inputs as strings. Hence, type conversion is required. For Example:
  ⟩⟩⟩ str(123)
  '123'
  ⟩⟩⟩ float(123)
  123.0
  ⟩⟩⟩ int(123.0)
  123
  ⟩⟩⟩ str(123.45)
  '123.45'
  ⟩⟩⟩ float('123.45')
  123.45
  ⟩⟩⟩ int('123.45') //String incompatible for conversion
  Traceback (most recent call last):
  File "⟨*pyshell*#3⟩", line 1, in ⟨*module*⟩
  int('123.45')
  ValueError: invalid literal for int() with base 10: '123.45'

Built-in Functions (Cont.)

- The functions *max* and *min* are used to find maximum and minimum values respectively; can also operate on string values.
- The integer and floating point values are compatible for comparison; whereas numeric values cannot be compared with string values. For Example:

```
⟩⟩⟩ max(59,80,95.6,95.2)
95.6
⟩⟩⟩ min(59,80,95.6,95.2)
59
⟩⟩⟩ max('hello', 'how', 'are', 'you')
'you'
⟩⟩⟩ min('hello', 'how', 'are', 'you', 'Sir')
'Sir'
```

- The function pow(a,b) computes a to the power b.

```
⟩⟩⟩ a = pow(2,3)
⟩⟩⟩ a
8
```

Built-in Functions (Cont.)

- The function random is used to generate a random number in the range [0,1). Python module random contains this function and needs to be imported for using it.
- Let us agree that player A will play the first turn if the generated falls in the range [0,0.5), otherwise player B will play as:

```
if random.random() < 0.5:
    print('Player A plays the first turn.')
else:

    print('Player B plays the first turn.')
```

- The math module provides some functions for mathematical computations.
- In order to obtain these functions available for use in script, we need to import the math module as:
  import math
- Name of the module, followed by the separator dot, should precede function name. The math module also defines a constant math.pi having value 3.141592653589793.

Built-in Functions (Cont.)

- Functions of math module are:

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
floor(x)	Returns the largest integer less than or equal to x.
fabs(x)	Returns the absolute value of x.
exp(x)	Returns the value of expression e**x.
log(x,b)	Returns the log(x) to the base b.In the case of absence of the second argument,the logarithmic value of x to the base e is returned.
log10(x)	Returns log(x) to the base 10.This is equivalent to specifying math.log(x,10).
pow(x,y)	Returns x raised to the power of y,i.e.,x**y.
sqrt(x)	Returns the square root of x.
cos(x)	Returns the cosine of x radians.
sin(x)	Returns the sine of x radians.
tan(x)	Returns the tangent of x radians.

Built-in Functions (Cont.)

- Functions of math module are (Cont.:

Function	Description
acos(x)	Returns the inverse cosine of x iin radians.
asin(x)	Returns the inverse sine of x in radians.
atan(x)	Returns the inverse tangent of x radians.
degrees(x)	Returns a value in degree equivalent of input value x(in radians).
radians(x)	Returns a value in radiant equivalent of input value x(in degrees).

- For Example:

```
⟩⟩⟩ import math
⟩⟩⟩ math.ceil(3.4)
4
⟩⟩⟩ math.floor(3.7)
3
```

Built-in Functions (Cont.)

- If we want to see the complete list the complete list of built-in functions, we can use the built-in function dir as
  dir(__builtins__)
- We can get the help on a function as:
  ⟩⟩⟩ import math
  ⟩⟩⟩ help(math.cos)
  Help on built-in function cos in module math:
  cos(...)
  cos(x)
  Return the cosine of x (measured in radians).

Function Definition and Call

- The syntax for function definition as:
  def function_name (comma_separated_list_of_parameters):

      statements
- For Example, We can print a triangle, a blank line and a square as:

```
def main():
    # To print a triangle
    print(' *')
    print(' ***')
    print(' *****')
    print('*******')
    # To print a blank line
    print()
    # To print a square
    print('* * * *')
    print('* * * *')
    print('* * * *')

    print('* * * *')
```

- Python code following colon must be intended, i.e., shifted right.

Function Definition and Call (Cont.)

- Having developed the function main in the script picture, we would like to execute it.
- To do this, we need to invoke the function main in the following two steps.
  1. In Run menu, click on the option Run Module.
  2. Using Python shell, invoke (call) the function main by executing the following command:

  ⟩⟩⟩ main()
- We can eliminate the need to call function main explicitly from the shell, by including in the script picture, the following call to function main:

```
if '__name__' == '__main__':
        main()
```

- Python module has a built-in variable called __name__ containing the name of the module. When the module itself is being run as the script, this variable __name__ is assigned the string '__main__' designating it to be a __main__ module.

Function Definition and Call (Cont.)

- Program to print a triangle followed by a square as:

```
def main():
        # To print a triangle
        print(' *')
        print(' ***')
        print(' *****')
        print('*******')
        # To print a blank line
        print()
        # To print a square
        print('* * * *')
        print('* * * *')
        print('* * * *')
        print('* * * *')
if '__name__' == '__main__':
        main()
```

Function main serves as a caller function for the callee or called functions triangle and square.

Function Definition and Call (Cont.)

- A function definition has no effect unless it is invoked.

```
def triangle():
        # To print a triangle
        print(' *')
        print(' ***')
        print(' *****')
        print('*******')
def main():
        # To print a triangle
        print('Triangle')
if '__name__' == '__main__':

        main()
```

The script shows that function triangle is not being invoked.

Function Definition and Call (Cont.)

- **return** statement returns the value of expression following the return keyword. In the absence of the return statement, default value None is returned.
- Arguments: Variables or Expressions whose values are passed to called function.
- Parameters: Variables or Expressions in function definition which receives value when the function is invoked.
- Arguments must appear in the same order as that of parameters.
- For Example, computing area of a rectangle:

```
def areaRectangle(length, breadth):
        '''
        Objective: To compute the area of rectangle
        Input Parameters: length, breadth- numerical value
        Return Value: area - numerical value
        '''
        area = length * breadth
        return area
```

Function Definition and Call (Cont.)

Fruitful Functions vs void Functions

- A function that returns a value is often called a fruitful function.
- A function that does not return a value is often called a void function.

Function Help

- Recall that function help can be used to provide a description of built-in functions.
- Function help can also be used to provide description of the function defined by user.
- Function help retrieves first multi-line comment from the function definition

Function Definition and Call (Cont.)

Default Parameter Values

- The function parameters may be assigned initial values also called **default values**.
- Function call uses default value, if value for a parameter is not provided.
- Non-default arguments should not follow default arguments in a function definition.For Example:

```
def areaRectangle(length, breadth=1):
        '''
        Purpose : To compute area of rectangle
        Input Parameters :
            length - int
            breadth (default = 1) - int
        Return Value: area - int
        '''
        area = length * breadth

        return area
>>> areaRectangle(5)
5
>>> areaRectangle(5,2)
10
```

Function Definition and Call (Cont.)

Keyword Arguments

- Python allows us to specify arguments in an arbitrary order in a function call, by including the parameter names along with arguments.
- The syntax for keyword arguments is:
  parameter_name = value
- indeed, in situations involving a large number of parameters, several of which may have default values, keyword arguments can be of great help. For Example:
  ⟩⟩⟩ def f(a=2, b=3, c=4, d=5, e=6, f=7, g=8, h=9):
  return a+b+c+d+e+f+g+h
  ⟩⟩⟩ f(c=10, g=20)
  62

Importing User-Defined Module

- To access a function from a user-defined module (also known as program or script that may comprise functions, classes, and variables), we need to import it from that module.
- To ensure that the module is accessible to the script, we are currently working on, we append to the system's path, the path to the the folder containing the module.
- The syntax for importing a module:
  import *name-of-the-module*

Assert Statement

- The assert statement is used for error checking.
- For Example, When we are going for the calculation of average of marks, be sure that inputs provided by the user are in the correct range.
- For this purpose, we make use of an assert statement that has the following form:
  assert *condition*
- If the condition specified in an assert statement fails to hold, Python responds with an assertion error.

References

Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Control Structures

Lecture 3

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

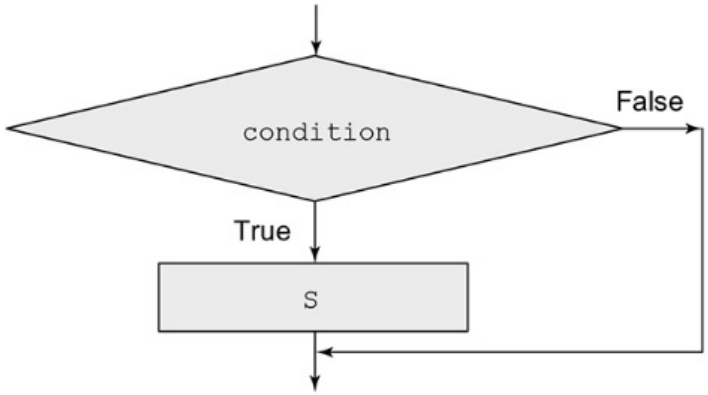
- Control structures are used for non-sequential and repetitive execution of instructions.
- Control statements (also called control structures) are used to control the flow of program execution by allowing non-sequential or repetitive execution of instructions.
- Python supports ***if, for, and while*** control structures.
- In a control statement, Python code following the colon (:) is indented.

if Conditional Statement

- *if* statement allows non-sequential execution depending upon whether the condition is satisfied.
- The general form of *if* conditional statement is as follows:
  if ⟨*condition*⟩ :

  Sequence S of statements to be executed

  Here, condition is a Boolean expression which is evaluated at the beginning of the if statement. If condition evaluates to True, then the sequence S of statements is executed, and the control is transferred to the statement following if statement. However, if condition evaluates to False, then the sequence S of statements is ignored, and the control is immediately transferred to the statement.



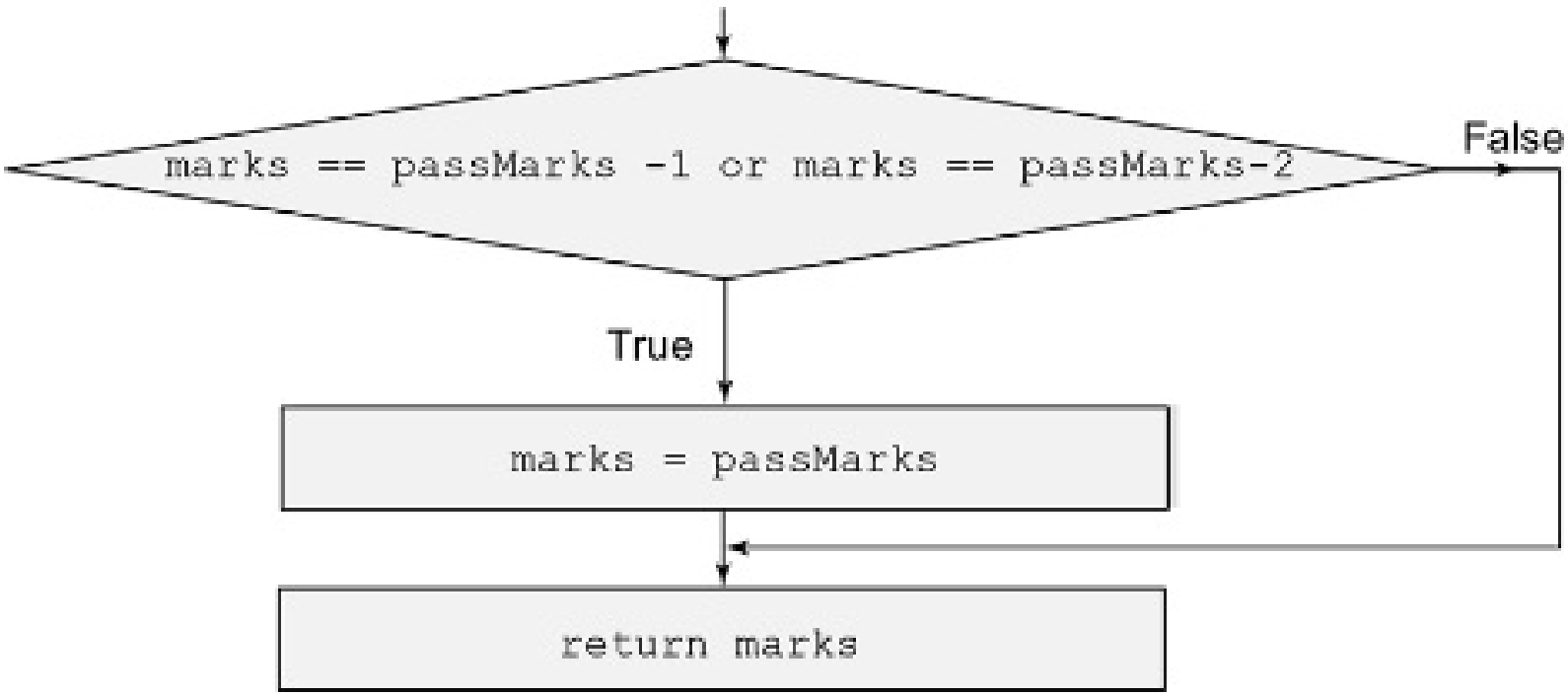
if Conditional Statement

- For Example (Marks to be updated to passMarks based on a condition):

```
def moderate(marks, passMarks):
    '''
    Objective to moderate result by maximum 1 or 2 marks to
    acheive passMarks
    Input parameters:
        marks - int
        passMarks - int
    Return Value: marks - int
    '''
    if marks == passMarks-1 or marks == passMarks-2:
        marks=passMarks
    return marks

def main():
    '''
    Objective: To moderate marks if a student just misses
    pass marks
    Input Parameter: None
    Return Value: None
    '''
    passMarks=40
    marks = input('Enter marks:')
    intmarks = int(marks)
    moderatedMarks = moderate(intmarks,passMarks)
    print('Moderated marks:',moderatedMarks)

if__name__='__main__':
    main()
```

if Conditional Statement

- For Example:
  ⟩⟩⟩
  Enter marks: 38
  Moderated marks: 40
- The flowchart of function moderate is as:

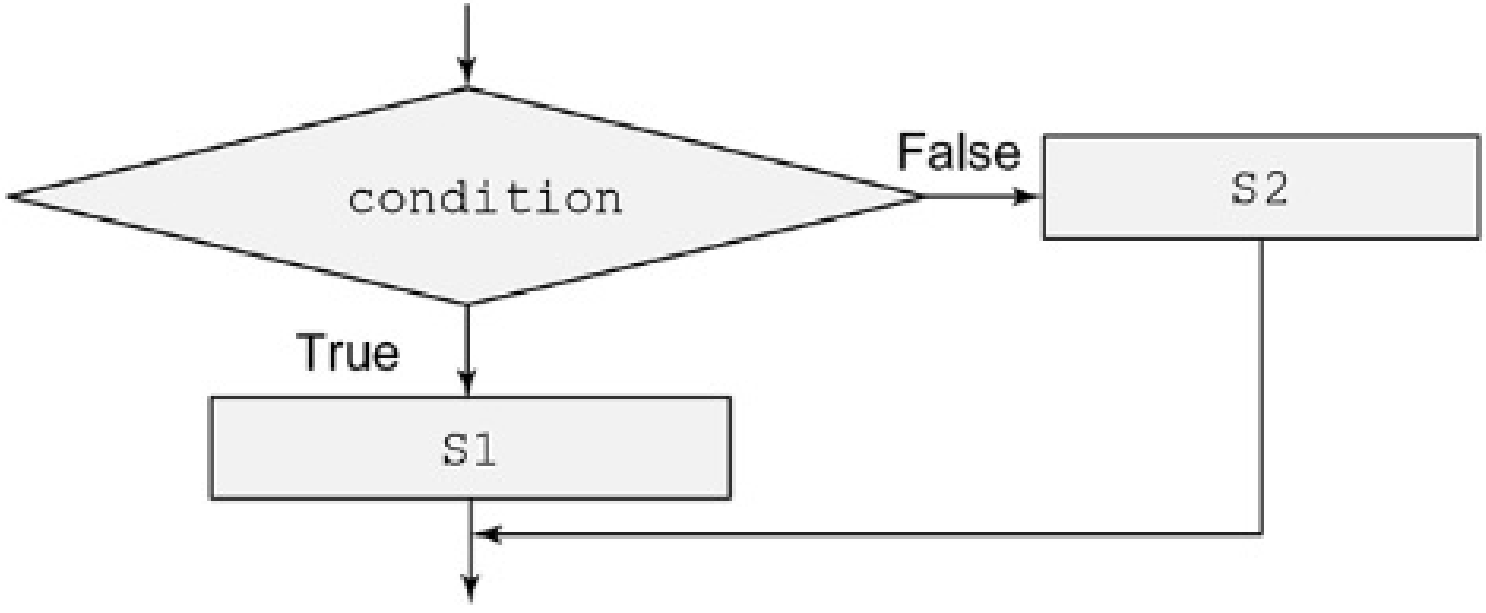


if Conditional Statement

- The general form of if-else statement is as follows:

if ⟨*condition*⟩ :
    Sequence S1 of statements to be executed
else:
    Sequence S2 of statements to be executed

Here, condition is a Boolean expression. If condition evaluates to True, then the sequence S1 of statements gets executed, otherwise, the sequence S2 of statements gets executed.



if Conditional Statement

- For Example (Program to authenticate user and allow access to system using if else statement):

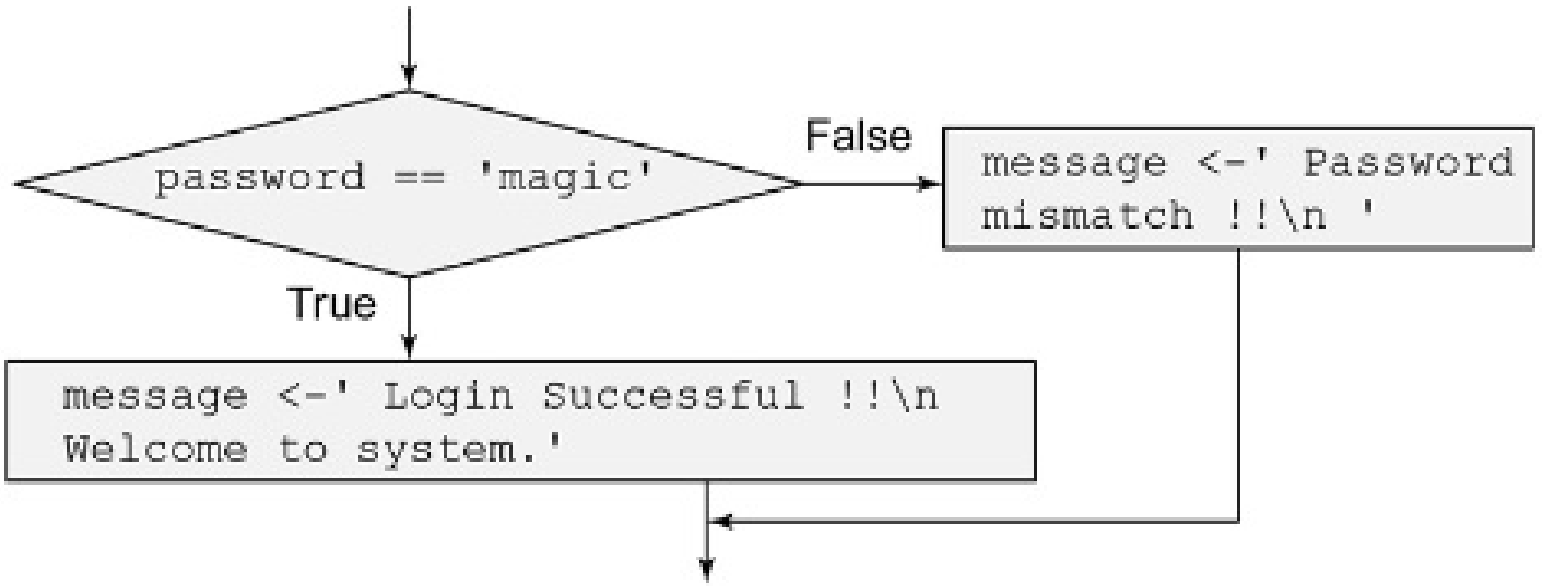
```
def authenticateUser(password):
    ""
    Objective to authenticate user and allow access to system
    Input parameter : password - string
    Return Value: message - string
    ""
    if password == 'magic':
        message='Login Successful!!\n Welcome to system.'
    if password != 'magic':
        message=' Password mismatch!!\n'
    return message

def main():
    ""
    Objective: To authenticate user
    Input Parameter: None
    Return Value: None
    ""
    print(' \t LOGIN SYSTEM ')
    print('======================')
    password = input('Enter Password:')
    intmarks = int(marks)
    message = authenticateUser(password)
    print(message)

if__name__='__main__':
    main()
```

if Conditional Statement

- For Example:

```
⟩⟩⟩
if password = 'magic':
        message = 'Login Successful !! \n'
else:
        message = 'Password mismatch !! \n'
```

- The flowchart of if-else statement in the function authenticate User is as:



if Conditional Statement

- The general form of if-elif-else statement is as follows:

if ⟨*condition*⟩ :
    Sequence S1 of statements to be executed
elif ⟨*condition*2⟩ :
    Sequence S2 of statements to be executed
elif ⟨*condition*3⟩ :
    Sequence S1 of statements to be executed
...
else:
    Sequence Sn of statements to be executed

The clauses elif and else of if control structure are optional. When a control structure is specified within another control structure, it is called nesting of control structures.

Iteration (for and while statements)

- The process of repetitive execution of statement or a sequence of statements is called a loop.
- Execution of a sequence of statements in a loop is known as an iteration of the loop.
- The control statement *for* is used when we want to execute a sequence of statements a fixed number of times.
- The general form of for statement is as follows:
  for variable in sequence :
  Sequence S of Statements
  Here, variable refers to the control variable. The sequence S of statements is executed for each value in sequence.
- The function call range(1, n+1) generates a sequence of numbers from 1 to n. In general, **range(start, end, increment)** produces a sequence of numbers from start up to end (but not including end) in steps of increment. If third argument is not specified, increment is assumed to be 1.

Iteration (for and while statements)

- For Example (Program to find sum of first n positive integers):

```
def summation(n):
    '''
    Objective: To find sum of first n positive integers
    Input Parameter : n - numeric value
    Return Value: total - numeric value
    '''
    total = 0
    for count in range(1,n+1):
        total += count
    return total

def main():
    '''
    Objective: To find sum of first n positive integers based on
    user input
    Input Parameter: None
    Return Value: None
    '''
    n = int(input('Enter number of terms:'))
    total = summation(n)
    print('Sum of first', n, 'positive integers:', total )

if__name__='__main__':
    main()
```

Iteration (for and while statements)

- A while loop is used for iteratively executing a sequence of statements again and again on the basis of a test-condition.
- The general form of while loop is as follows:

while $\langle test - condition \rangle$:

Sequence S of statements

Here, test-condition is a Boolean expression which is evaluated at the beginning of the loop. If the test-condition evaluates to True, the control flows through the Sequence S of statements (i.e, the body of the loop), otherwise the control moves to the statement immediately following the while loop. On execution of the body of the loop, the test-condition is evaluated again, and the process of evaluating the test-condition and executing the body of the loop is continued until the test-condition evaluates to False.

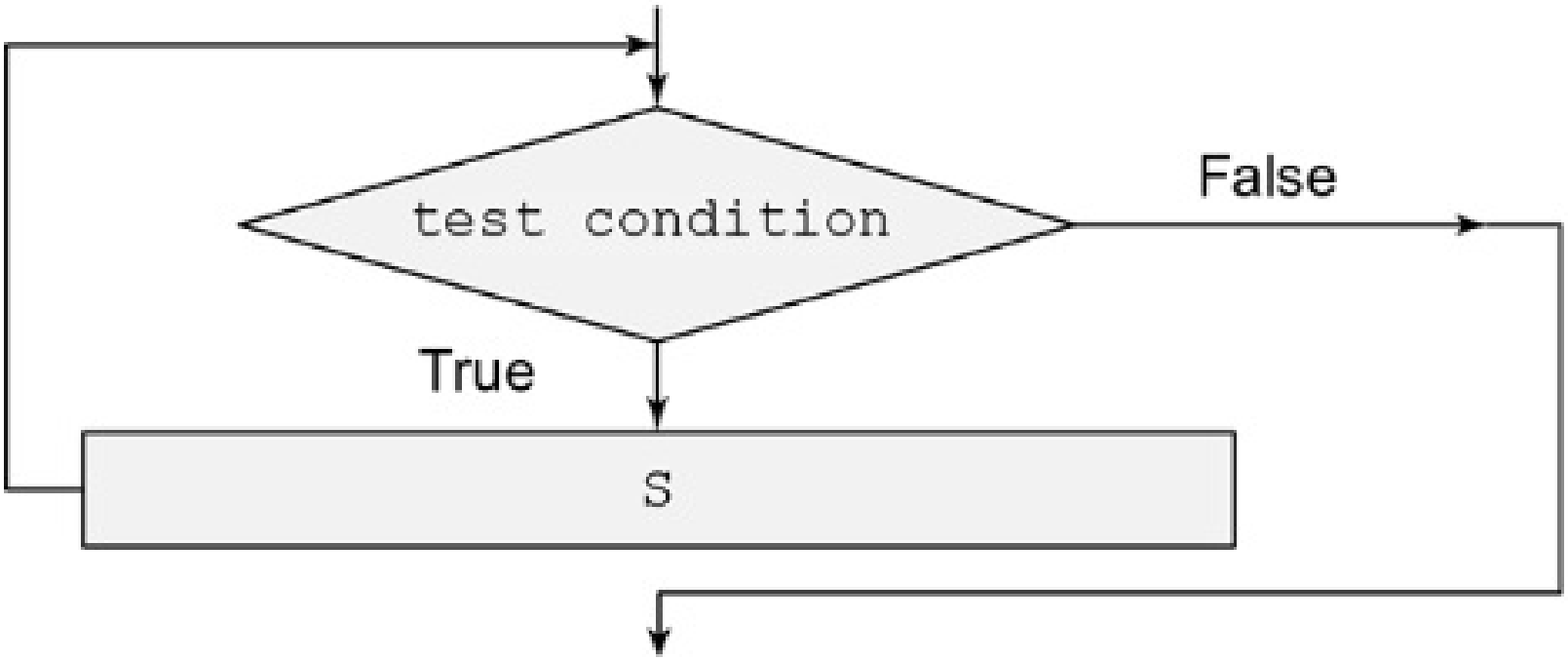
Iteration (for and while statements)

- For Example (Program to compute sum of numbers):

```
def main():
        '''
        Objective: To compute sum of numbers entered by user until
        user provides with null string as the input
        Input Parameter: None
        Return Value: None
        '''
        total=0
        number = input('Enter a number')
        while number != ' ':
            total += int(number)
            number = input('Enter a number: ')
        print('Sum of all input numbers is', total)

if__name__='__main__':
        main()
```

Iteration (for and while statements)

- The flow diagram of While structure:



Iteration (for and while statements)

while statement vs for statement

- Let us consider the code of for statement:

```
for count in range(1, n+1):
        total += count
```

- We can re-write the above statement in while statement:

```
count = 1
while count < n+1:
        total += count

        count += 1
```

- N.T. - For computing the sum of first few natural numbers, the use of for loop is more elegant and easy as compared to the while loop.

Iteration (for and while statements)

Example: To print some pictures

- Program to print right triangle and inverted triangle:

```
def rightTriangle(nRows):
        '''
        Objective: To print right triangle
        Input Parameter: nRows - integer value
        Return Value: None
        '''
        for i in range(1, nRows + 1)
            print(' * ' * i)

def invertedTriangle(nRows):
        '''
        Objective: To print inverted triangle
        Input Parameter: nRows - integer value
        Return Value: None
        '''
        nSpaces = 0
        nStars = 2 * nRows - 1
        for i in range(1, nRows+1):
            print(' ' * nSpaces + ' * ' * nStars)
            nStars -=2
            nSpaces +=1
```

Iteration (for and while statements)

Example: To print some pictures [Cont.]

```
def main():
    '''
    Objective: To print right triangle or inverted triangle
    depending on user 's choice
    Input Parameter: None
    Return Value: None
    '''
    choice = int(input('Enter 1 for right triangle.\n'+ 'Enter 2 for inverted triangle.\n'))
    assert choice == 1 or choice == 2
    nRows = int(input('Enter no. of rows))
    if choice == 1:
        rightTriangle(nRows)
    else:
        invertedTriangle(nRows)
if__name__='__main__':
    main()
```

Output: (a) Right Triangle

```
*
* *
* * *
* * * *
* * * * *
```

(b) Inverted Triangle

```
* * * * * * * * *
 * * * * * * *
  * * * * *
   * * *
    *
```

Iteration (for and while statements)

Examples of nested control structures

for(...):
  #Sequence of statements S1
  for(...):
  #Sequence of statements S2
  #Sequence of statements S3
	while (...):
	  #Sequence of statements S1
	  while(...):
	  #Sequence of statements S2
	  #Sequence of statements S3

for(...):
  #Sequence of statements S1
  while(...):
  #Sequence of statements S2
  #Sequence of statements S3
	while (...):
	  #Sequence of statements S1
	  for(...):
	  #Sequence of statements S2
	  #Sequence of statements S3

for(...):
  #Sequence of statements S1
  for(...):
  #Sequence of statements S2
  while(...):
  #Sequence of statements S3
  #Sequence of statements S4
  #Sequence of statements S5
	while (...):
	  #Sequence of statements S1
	  for(...):
	  #Sequence of statements S2
	  while(...):
	  #Sequence of statements S3
	  #Sequence of statements S4
	  #Sequence of statements S5

Iteration (for and while statements)

break, continue and pass statements

- The break statement is used for exiting from the loop to the statement following the body of the loop.
- The continue statement is used to transfer the control to next iteration of the loop without executing rest of the body of loop.
- The pass statement lets the program go through this piece of code without performing any action.
- The else clause can be used with for and while loop. Statements specified in else clause will be executed on smooth termination of the loop. However, if the loop is terminated using break statement, then the else clause is skipped.

References

[1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Debugging

Lecture 4

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- When a program fails to yield the desirable result, we say that it contains a bug.
- The bug could be an error such as division by zero, invalid type conversion, using a variable not defined, wrong initialization, or some other unintended operation being performed.
- The process of discovering the bugs and making the program bug-free is called debugging.
- debugging: making the program error free.

Testing

- Program testing aims to expose the presence of bugs in the programs.
- To find the bugs in a program, we test it on various inputs.
- We correct the errors found in this process and feel confident that the program will run smoothly.

Testing

- For Example (Finding Maximum of Three Numbers):

```
def max3(n1, n2, n3):
        '''
        Objective: To find maximum of three numbers
        Input Parameters: n1,n2,n3 - numeric values
        Return Value: maxNumber - numeric value
        '''
        maxNumber=0
        if n1< n2 :
                if n1 > n3:
                        maxNumber = n1
       elif n2 > n3:
       maxNumber=n2
       else:
                maxNumber = n3
        return maxNumber
def main():
        '''
       Objective: To find maximum of three numbers
       Input Parameter: None
       Return Value: None
        '''
       n1 = int(input('Enter first number: '))
       n2 = int(input('Enter second number: '))
       n1 = int(input('Enter third number: '))
       maximum = max3(n1,n2,n3)
       print('Maximum number is:',maximum)

if__name__='__main__':
       main()
```

Testing

- The function max3 is intended to return the maximum of three numbers.Since the input numbers can be either positive or negative,we may have test cases containing various combina tions of positive and negative numbers.
- We test this script on various permutations of the inputs: 10, 20, 30, namely:

1. 30, 20, 10	4. 20, 30, 10
2. 30, 10, 20	5. 10, 30, 20
3. 20, 10, 30	6. 10, 20, 30

- It so turns out that the result obtained is correct for all input permutations, except for the permutations 20, 10, and 30:

```
⟩⟩⟩
Enter first number: 20
Enter second number: 10
Enter third number: 30
Maximum number is 0
```

- Thus, a bug persists in the program that needs to be detected and removed.

Debugging

- There are various Python debugging tools such as pdb, pudb, pydb, and pydbgr. In this section, we will discuss Python's built-in debugger pdb, which is an interactive tool that helps in examining the code execution step by step.
- Debugging allows us to stop the execution of a program at a chosen instruction called a break point, and evaluate the various expressions in the current context.
- In order to debug a program in Python shell, we need to import the module pdb as well as the script to be debugged.
  ⟩⟩⟩ import pdb
  ⟩⟩⟩ import max3
  ⟩⟩⟩ pdb.run('max3.main()')
  > <string >(1) <module>()->None
- Python debugger uses the prompt (Pdb) to indicate that the program is executing in the debugging mode.

Debugging

- Another way to execute a program in debugging mode is by including the following two lines in the script:

```
import pdb
pdb.set_trace()
```

- Note that including above two lines at the beginning of the program for Finding maximum of three numbers. It will increase every line number by two. By invoking the function set_trace() at the very beginning, the program would run in debugging mode right from the first instruction to be executed.

- However, since we know that the bug exists in the function max3, we may invoke set_trace() within the function body.

Debugging

- Let us have a look at debugging commands:

Command	Explanation
h **or** help	it lists all the availble commands.
h commandName	it prints the description about a command.
w **or** where	Prints the stack trace (sequence of function calls currently in execution, most recent function call being at the beginning). Also shows the statement to be executed next.
u **or** up	Moves to the stack frame one level up in the stack trace
d **or** down	Moves to the stack frame one level down in the stack trace
s **or** step	Executes the current statement and moves the control to either next statement, or the function being invoked in the current statement.
n **or** next	Executes the current statement and moves the control to the next statement.unlike step command, if a function is being invoked in the current statement, the invoked function gets executed completely.
r **or** return	Continue execution until the current function returns.
p expression print(expression)	Prints the value of the specified expression in the current context.

Debugging

Command	Explanation
j (jump) lineno	Jumps to the given line number for the next statement to be executed.
l **or** list	List 11 lines in the vicinity of the current statement
b **or** break [[file :]] line[func[,cond]]	Sets the breakpoint at the specified line.Name of the file is one of the optional arguments. When the argument **func** is used, the breakpoint is set at the first executable statement of the specified function. The second argument may be used to denote a condition which must evaluate to **True** for setting the breakpoint.
tbreak [[file :]] line[func[,cond]]	Similar to **break** command. However, the break point being set is automatically removed once it is reached
cl **or** clear [[file :]] line[func[,cond]]	Clears the specified breakpoint. In the absence of an argument,clears all the breakpoints.
c **or** continue	Continue execution untill the breakpoint is reached
a **or** args	Prints the argument list of the current function along with their values.
q **or** quit	Quits from the python debugger

Debugging

- Execution of Finding Maximum of Three Numbers Program in debugging for the inputs 20,10,30 is shown below:

> f:\pythoncode \ch04 \max3.py<module>()
-> def max3(n1,n2,n3):
(pdb) h s
s(step)
Execute the current line, stop at the first possible
ocassion (either in a function that is called or in the current function)
(pdb)s
> f:\pythoncode \ch04 \max3.py<module>()
-> def main():
(pdb)s
> f:\pythoncode \ch04 \max3.py<module>()
->if__name__=’__main__’:
(pdb)s
> f:\pythoncode \ch04 \max3.py <module>()
->main()
(pdb)s
–call– > f:\pythoncode \ch04 \max3.py main()
->def main()

Debugging

(pdb)s
> f:\pythoncode \ch04 \max3.py main()
n1= int(input('Enter first number: '))
(pdb) n
Enter first number: 20
> f:\pythoncode \ch04 \max3.py main()
n1= int(input('Enter second number: '))
(pdb) n
Enter second number: 10
> f:\pythoncode \ch04 \max3.py main()
n1= int(input('Enter third number: '))
(pdb) n)
Enter third number: 30
> f:\pythoncode \ch04 \max3.py main()
->maximum = max3(n1,n2,n3)
(pdb) p (n1, n2, n3)
(20,10,30)

Debugging

(pdb) n
> f:\pythoncode \ch04 \max3.py main()
-> print('Maximum number is',maximum)
(pdb) n
Maximum number is 0
–Return–
> f:\pythoncode \ch04 \max3.py main() -< None
-> print('Maximum number is',maximum)
(pdb) s
–Return–
> f:\pythoncode \ch04 \max3.py <module>()-> None
-> main()
(pdb) q
Traceback (most recent call last):
File "F:/pythoncode/ch04/max3.py",27 line 32, in <module >
main()
bdb.BdbQuit

References

[1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Scope

Lecture 5

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University)
Bhubaneswar, Odisha, India

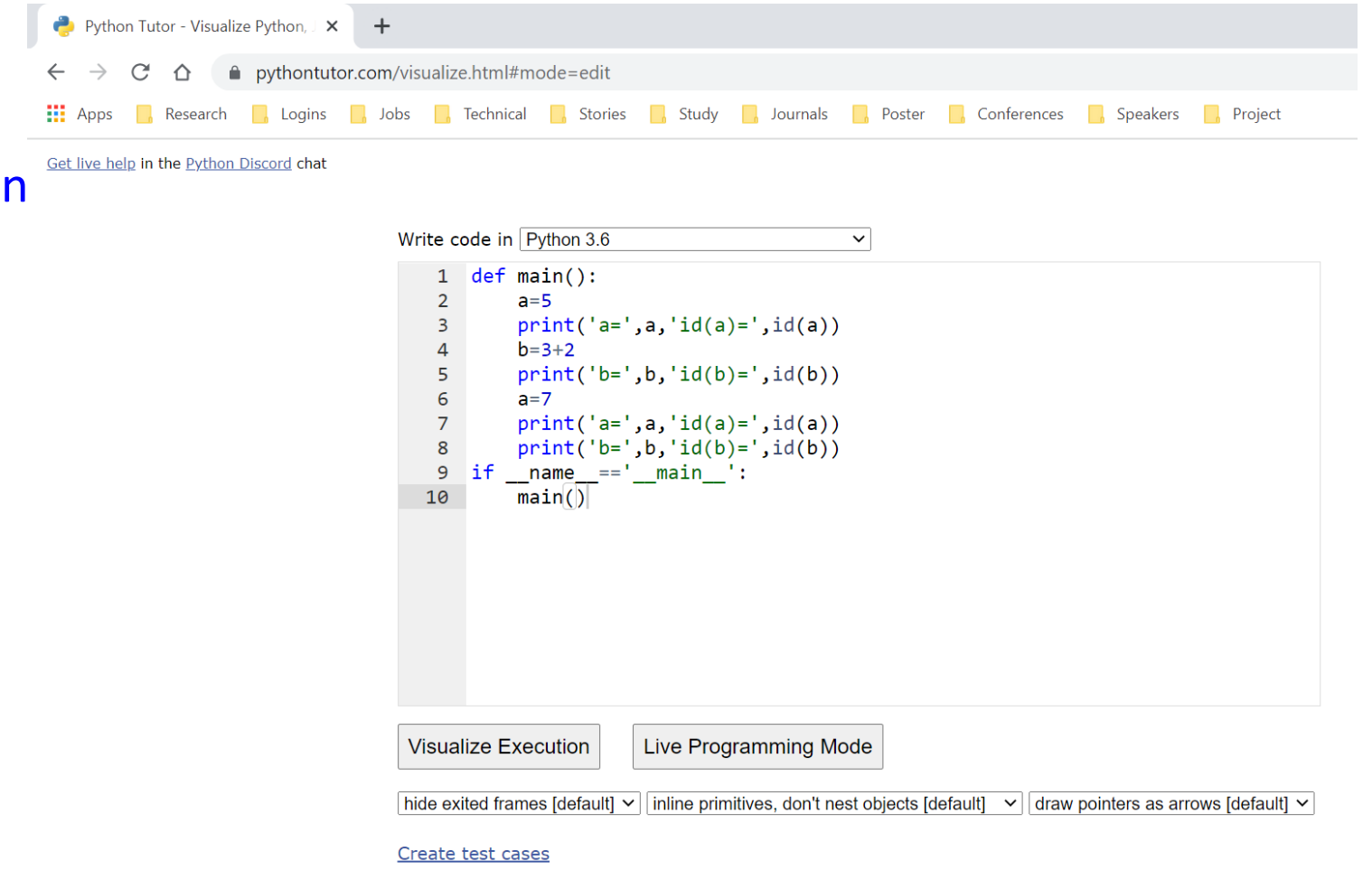
Contents

- Visualization in Python tutor
- Objects and Object IDs
- Namespaces
- Scope

*Python Programming: A Modular Approach by Sheetal Taneja and Kumar Naveen, Pearson Education India, Inc., 2017

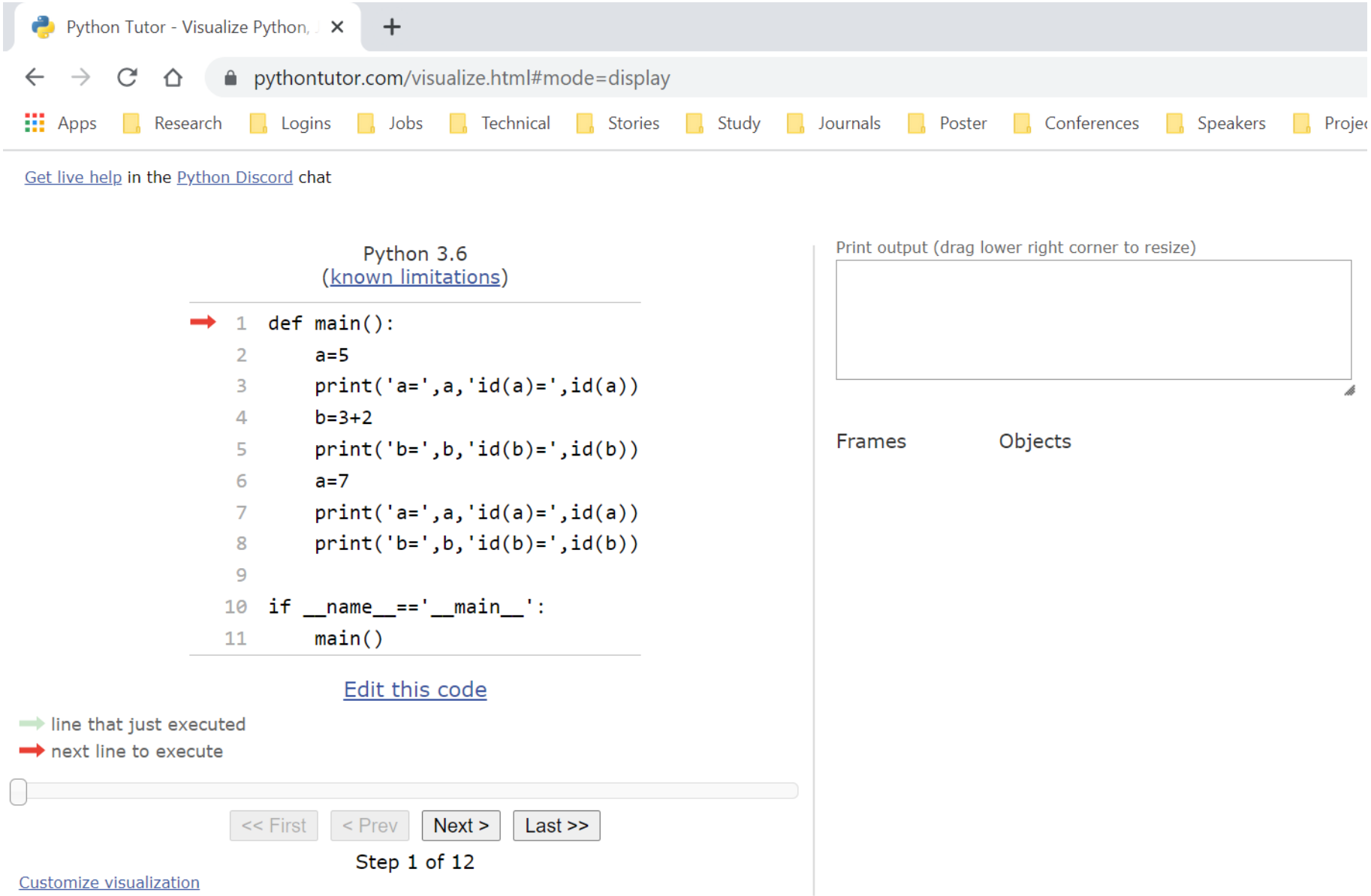
Visualization of Example-I in Python tutor

- Each object in Python is assigned a unique identifier that can be accessed using the function `id`
  - Visit the following website www.pythontutor.com
  - Click on Edit this code
  - Now write the code
  - Click on Visualize Execution



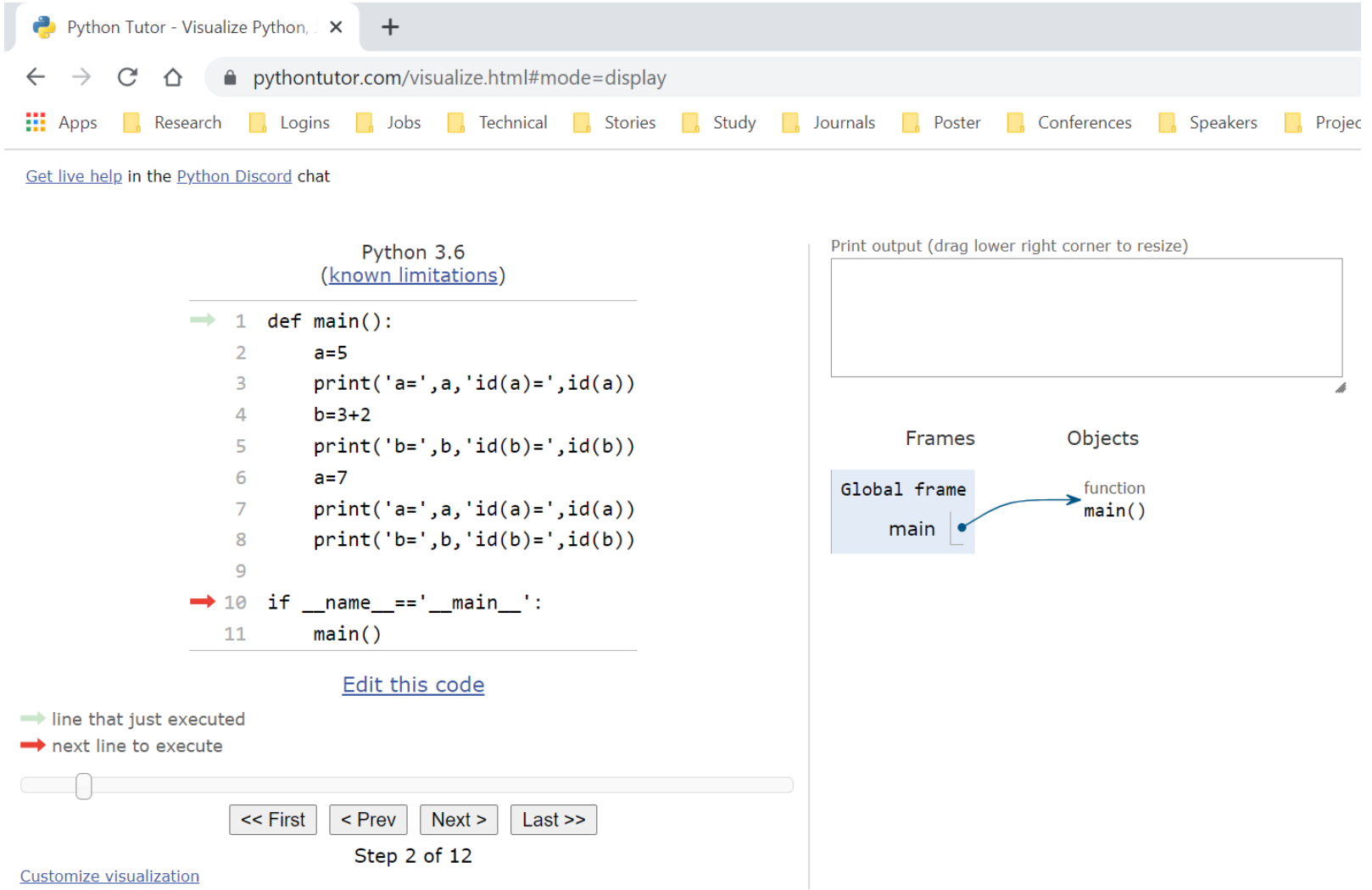
Example-I (continued)

- After clicking on Visualize Execution, a **<Print output>** box will appear at the right top



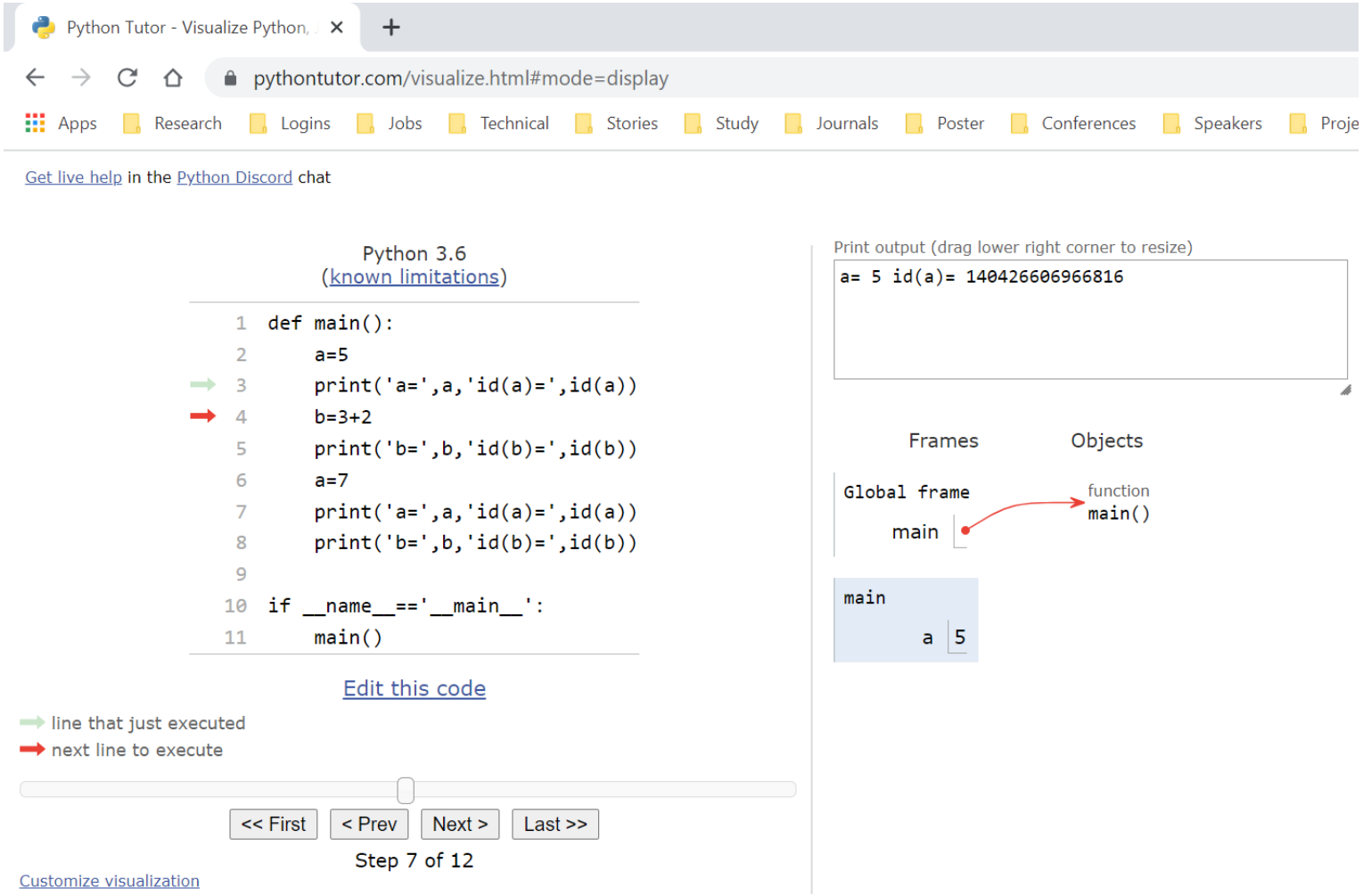
Example-I (continued)

- To start visualization, we click **<Next >**. On encountering the **`main`** function definition, the global frame lists the identifier **`main`** as shown in the Figure
- The red arrow marks the next line to be executed, and the green arrow marks the line just executed



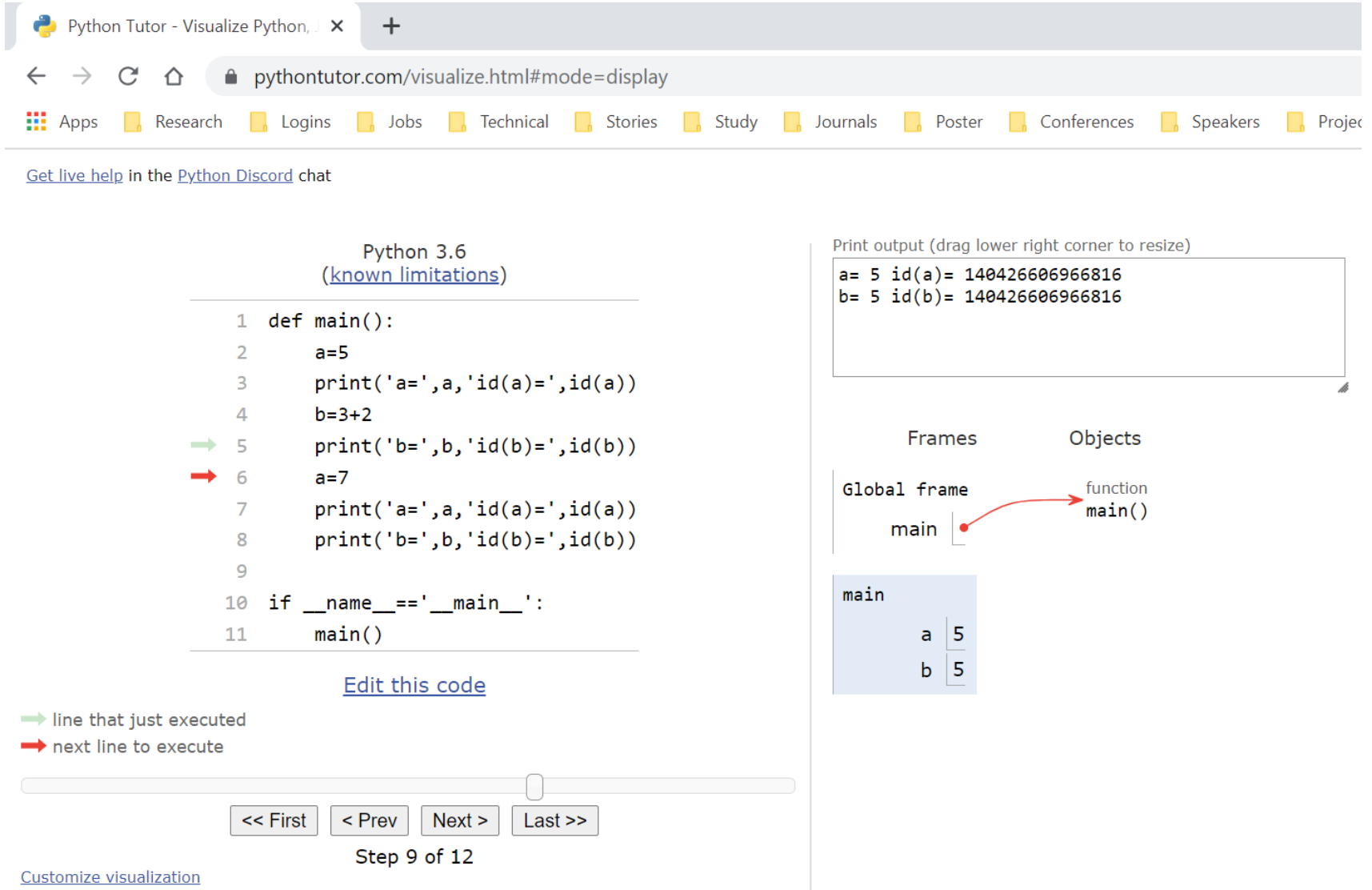
Example-I (continued)

- Now clicking **<Next>**, it executes the **`if`** statement. Clicking **<Next>** again executes the call to the function **`main`** and the visualizer shows the frame for the **`main`** function
- Clicking **<Next>**, it moves the next line pointer to line 2, and its execution shows the creation of **`int`** object **`5`** having name **`a`**. Later, clicking **<Next>** shows the output of the execution of line 3 in **<Print output>** box



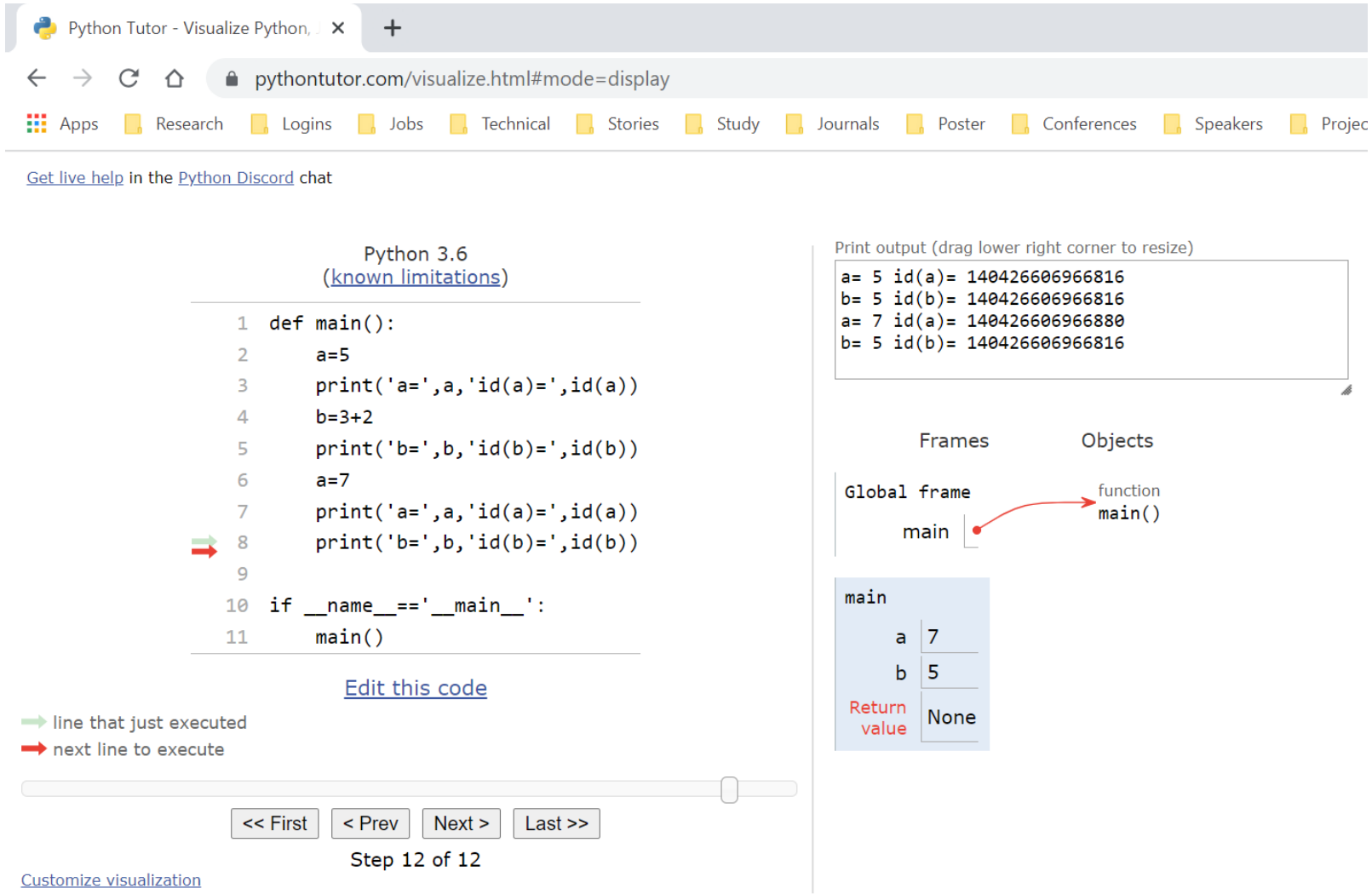
Example-I (continued)

- Next click executes line 4 and the name **b** is mapped to the **int** object **5** created earlier
- Further click executes line 5 and the output appears in the **<Print output>** box



Example-I (continued)

- Clicking **<Next>** executes line 6, resulting in creation of a new **`int`** object **`7`** and its mapping to **`a`**
- Further clicks execute lines 7 and 8 and the output appears in the <**Print output**> box
- Next click shows return value **`None`** associated with the function **`main`** as it does not return any value



Explanation of execution for Example-I

- Recall that when this program script is executed, Python makes a note of the definition of the function main in the global frame
- Next, on encountering the `if` statement, Python checks whether the name of the current module is `__main__`
- This being true, the expression `__name__ == '__main__'` evaluates as True, and the function main gets invoked
- Next, the statements in the main function are executed in a sequence
- Execution of line 2 creates an `int` object `5` and assigns it the name `a`. This object has `a` unique object id but can have multiple names as the execution of the script proceeds.
- For example, execution of the statement

```
b = 3 + 2
```

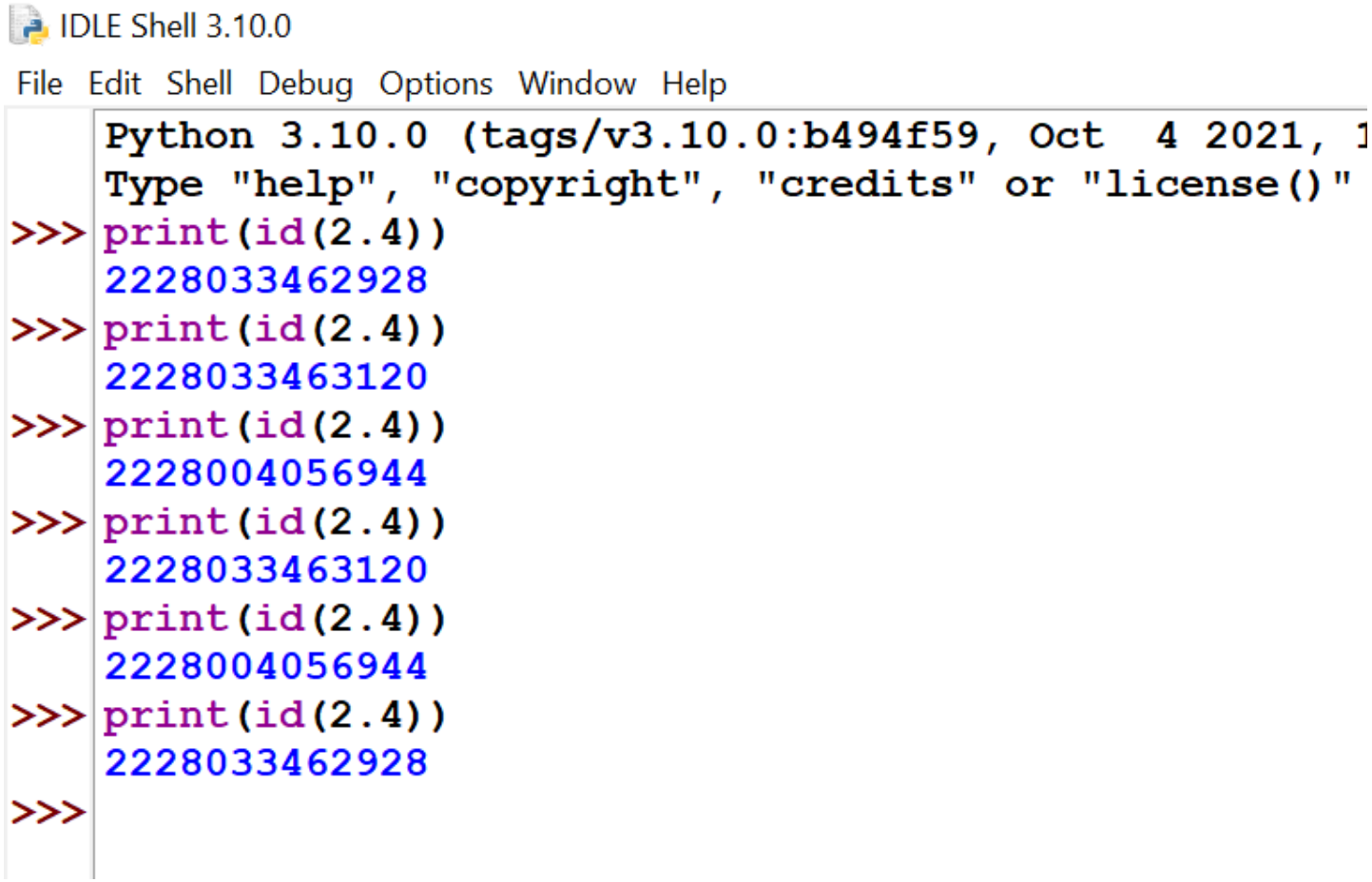
in line 4 does not generate a new object, but only associates the name `b` to the `int` object `5` created earlier

- Now, `a` and `b` have the same object id. However, execution of line 6 creates an `int` object `7` and associates it with the name `a`
- The name `b` continues to be associated with int object `5` created earlier.

*Python Programming: A Modular Approach by Sheetal Taneja and Kumar Naveen, Pearson Education India, Inc., 2017

Object ID for different data types

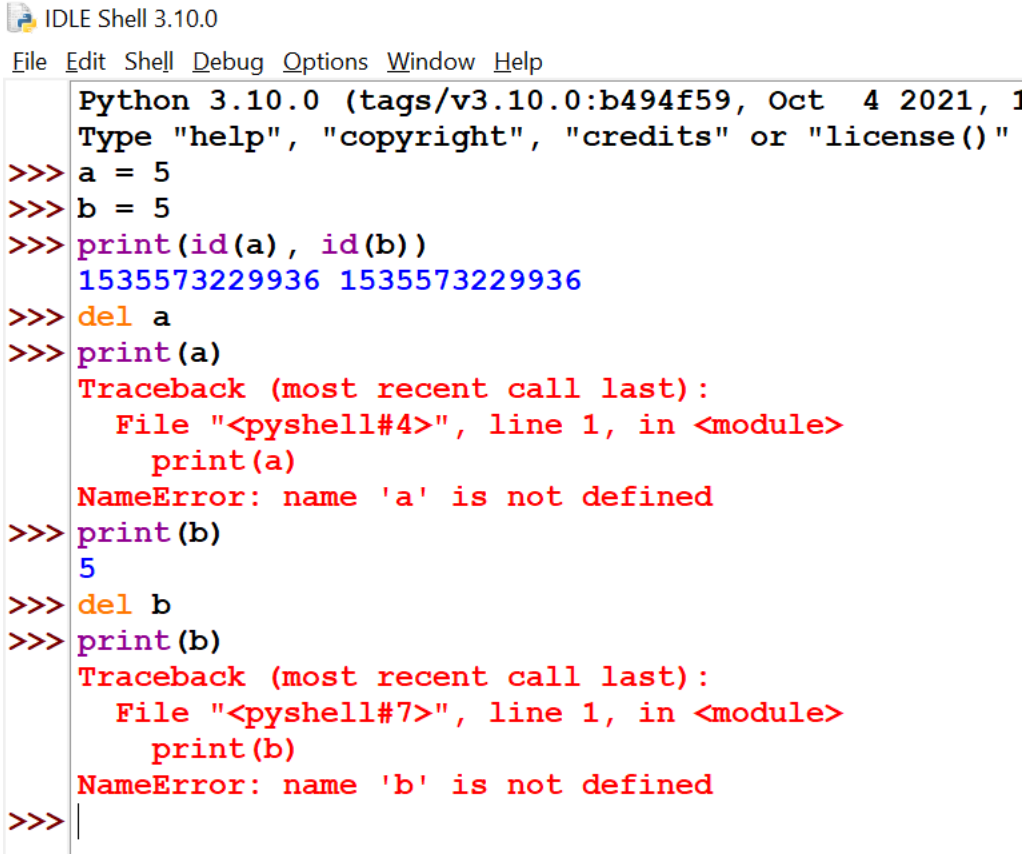
- The general principle is expressed by saying that Python caches or interns small integer objects (typically, up to 100) for future use. The same may not hold for other forms of data



- Note that the first three instructions create new objects
- However, subsequent instructions sometimes used the objects created earlier

del operator

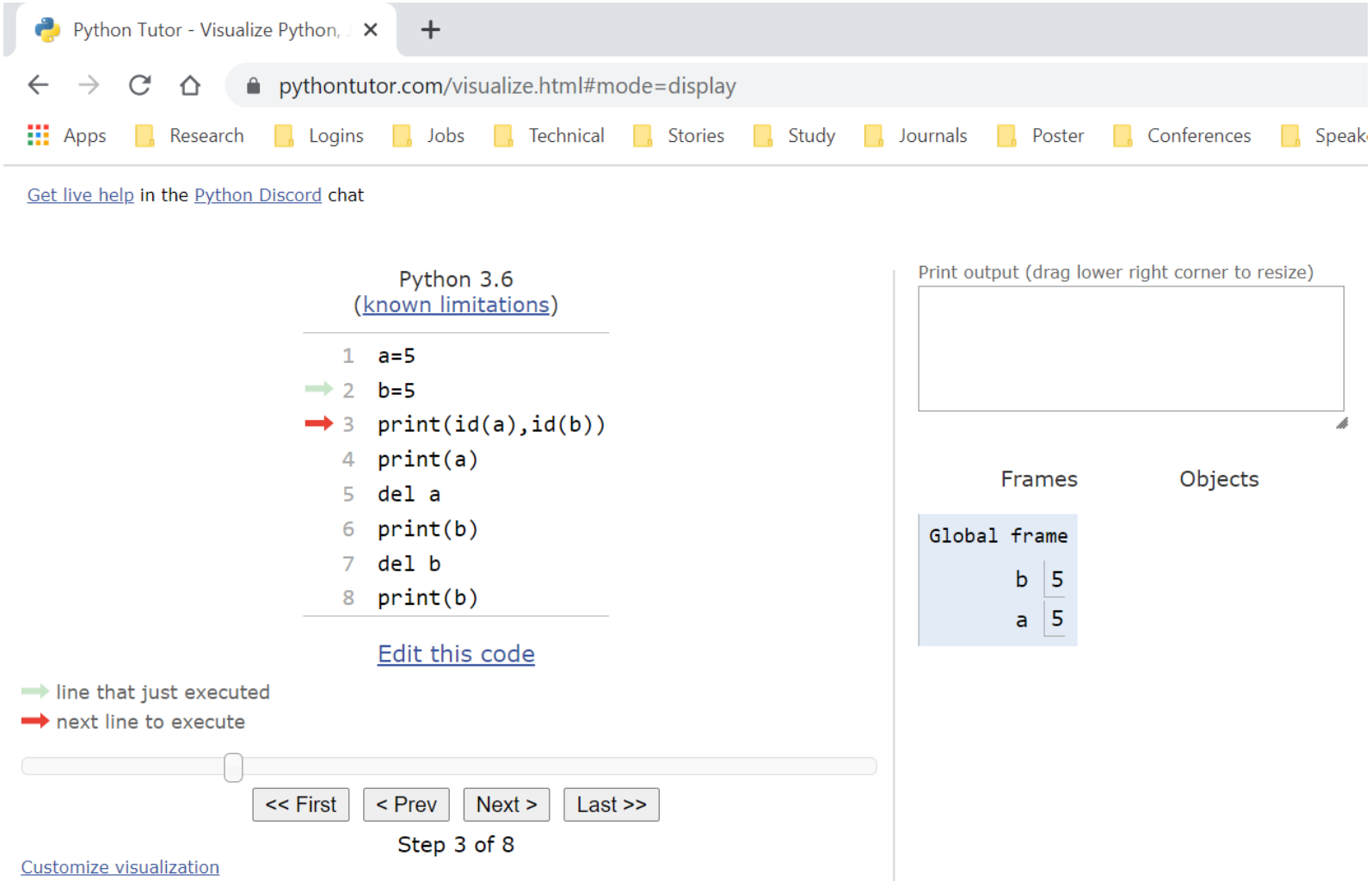
- It makes a name (i.e. the association between the name and the object) undefined



- The first statement creates an **`int`** object **`5`** and binds it to name **`a`**
- The second statement does not create a new object
- It binds the same object to name **`b`** and thus creates another reference to **`int`** object **`a`**

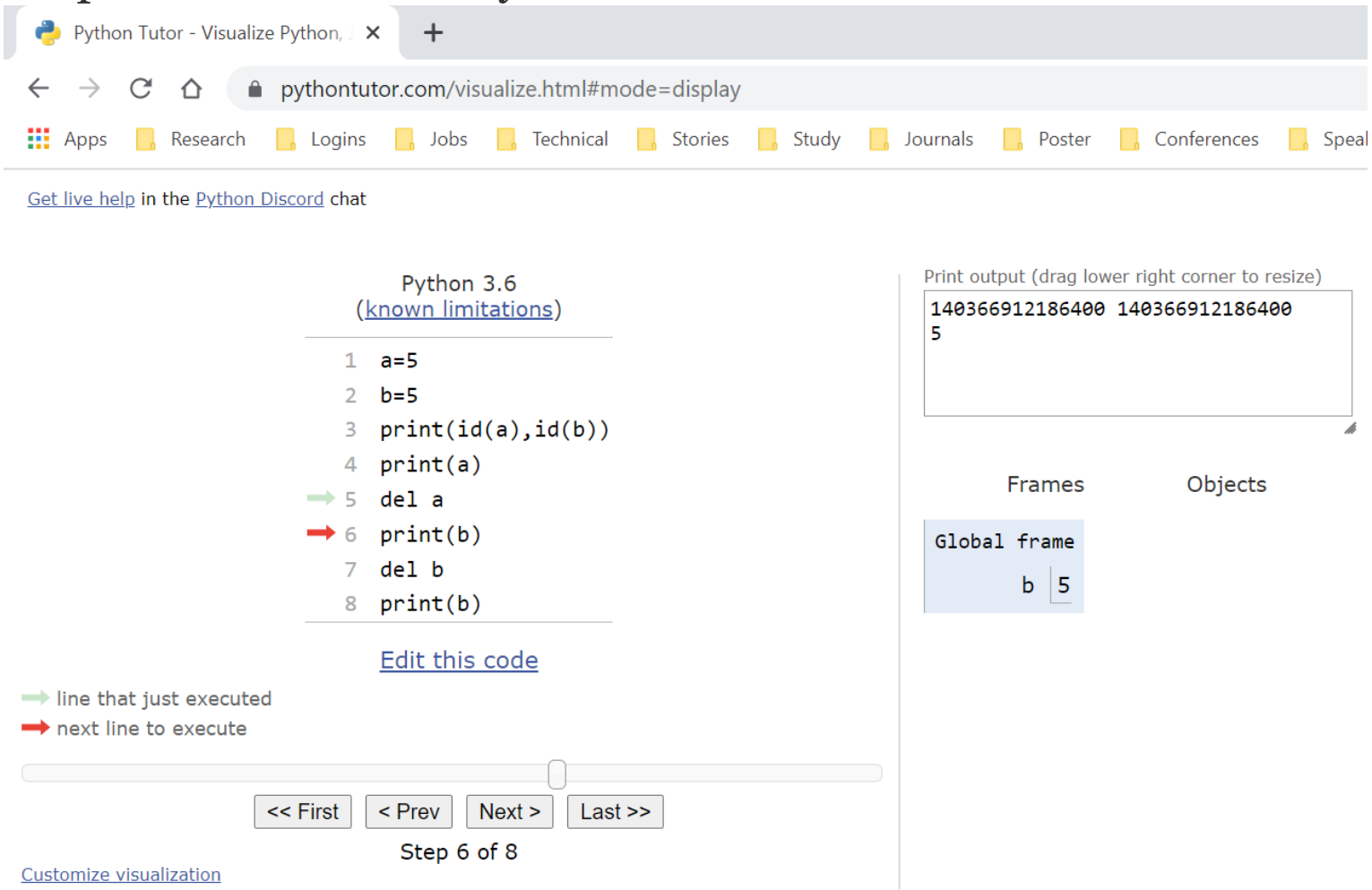
Visualization of `del` operator in Python tutor

- The first statement creates an `int` object `5` and binds it to name `a`
- The second statement does not create a new object
- It binds the same object to name `b` and thus creates another reference to `int` object `a`



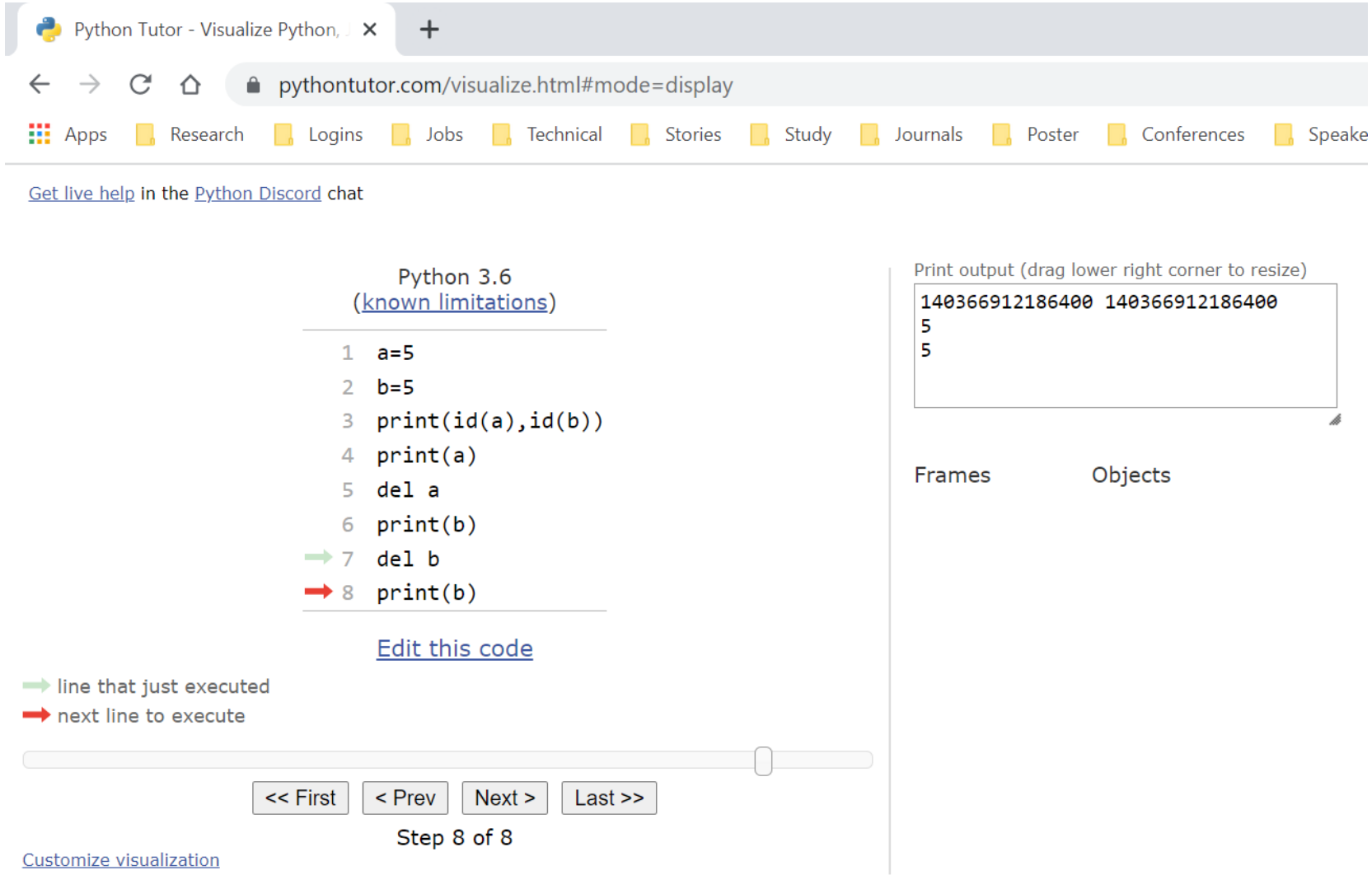
Visualization of `del` operator (continued)

- Python keeps a count of the number of references to an object
- When the statement `del a` is executed, it reduces the reference count of `int` object `5` from 2 to 1 and removes the binding of name `a` to `int` object `5`
- Thus, an attempt to access `a` now yields an error



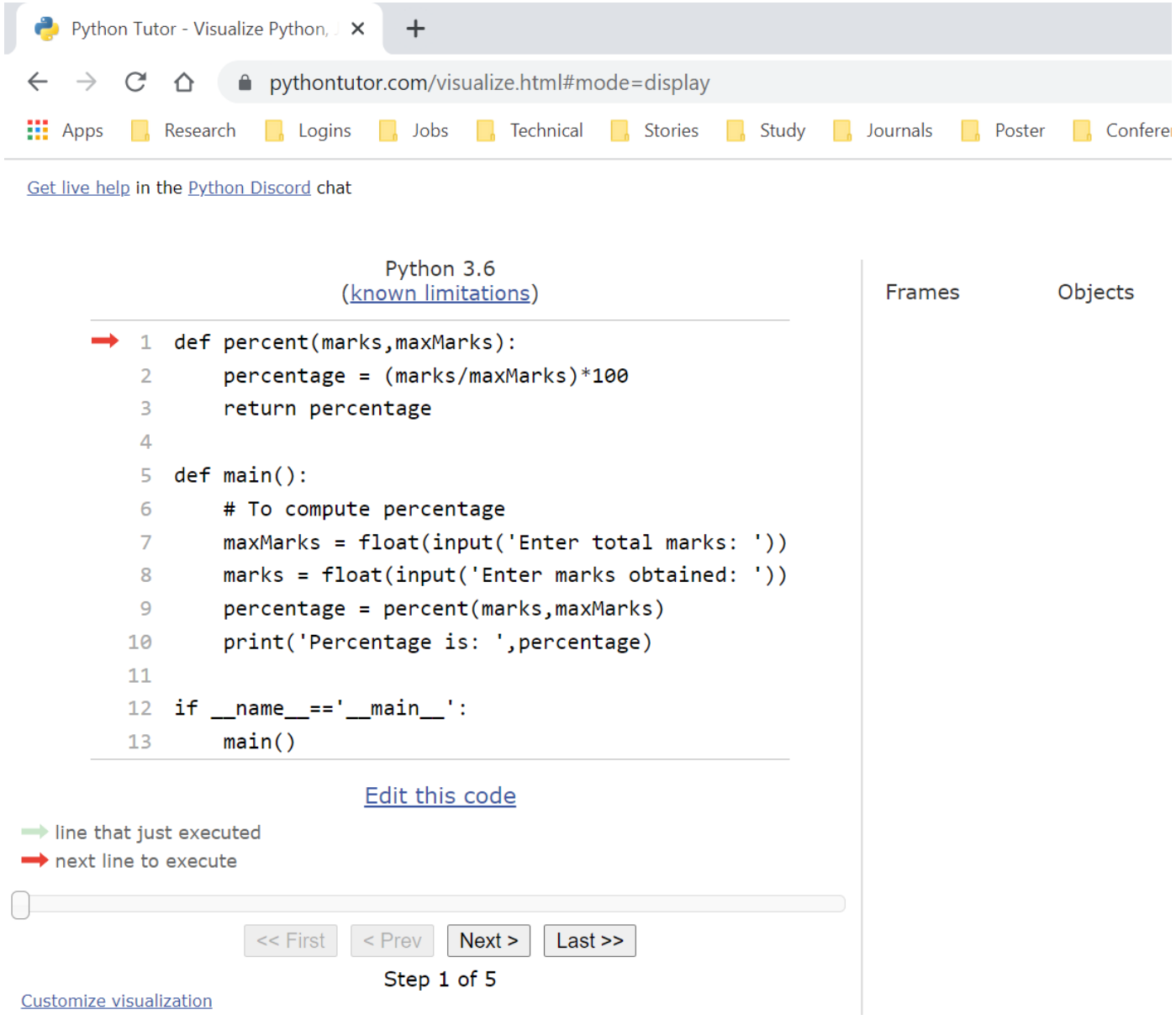
Visualization of `del` operator (continued)

- When the statement `del b` is executed, it reduces the reference count of `int` object `5` from `1` to `0`, and removes the binding of name `b` to `int` object `5`
- Thus, an attempt to access `b` now yields an error



Visualization of Example-II in Python tutor

- Write a program for calculating the percentage of a student in a subject

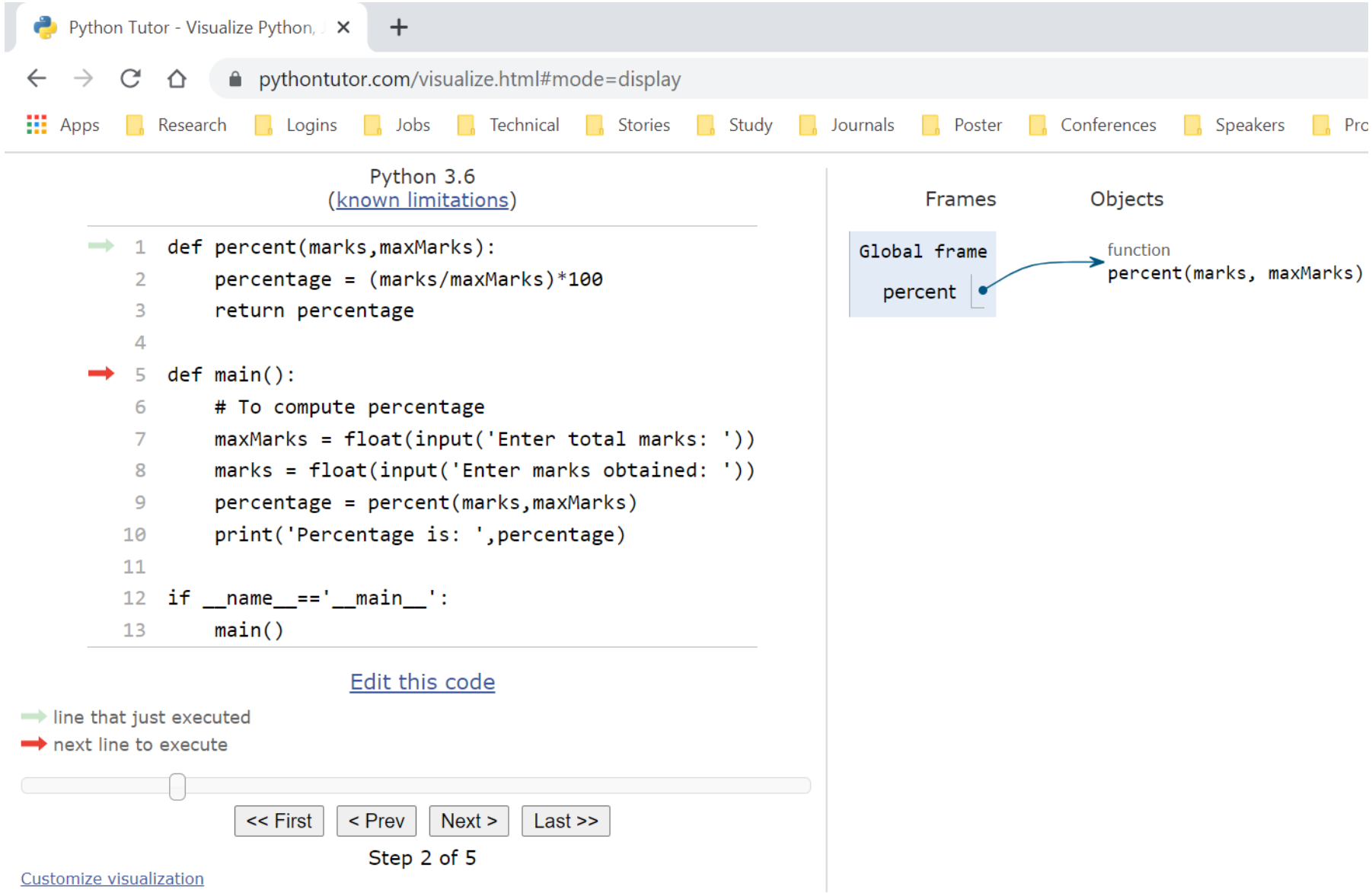


Visualization of Example-II (continued)

- We are about to execute the first of the five steps in the script
- These five steps are as follows:
  - Definition of function **`percent(marks, maxMarks)`**
  - Definition of function **`main()`**
  - **`if`** statement
  - Invoking the function **`main()`**
  - Execution of function **`main()`**

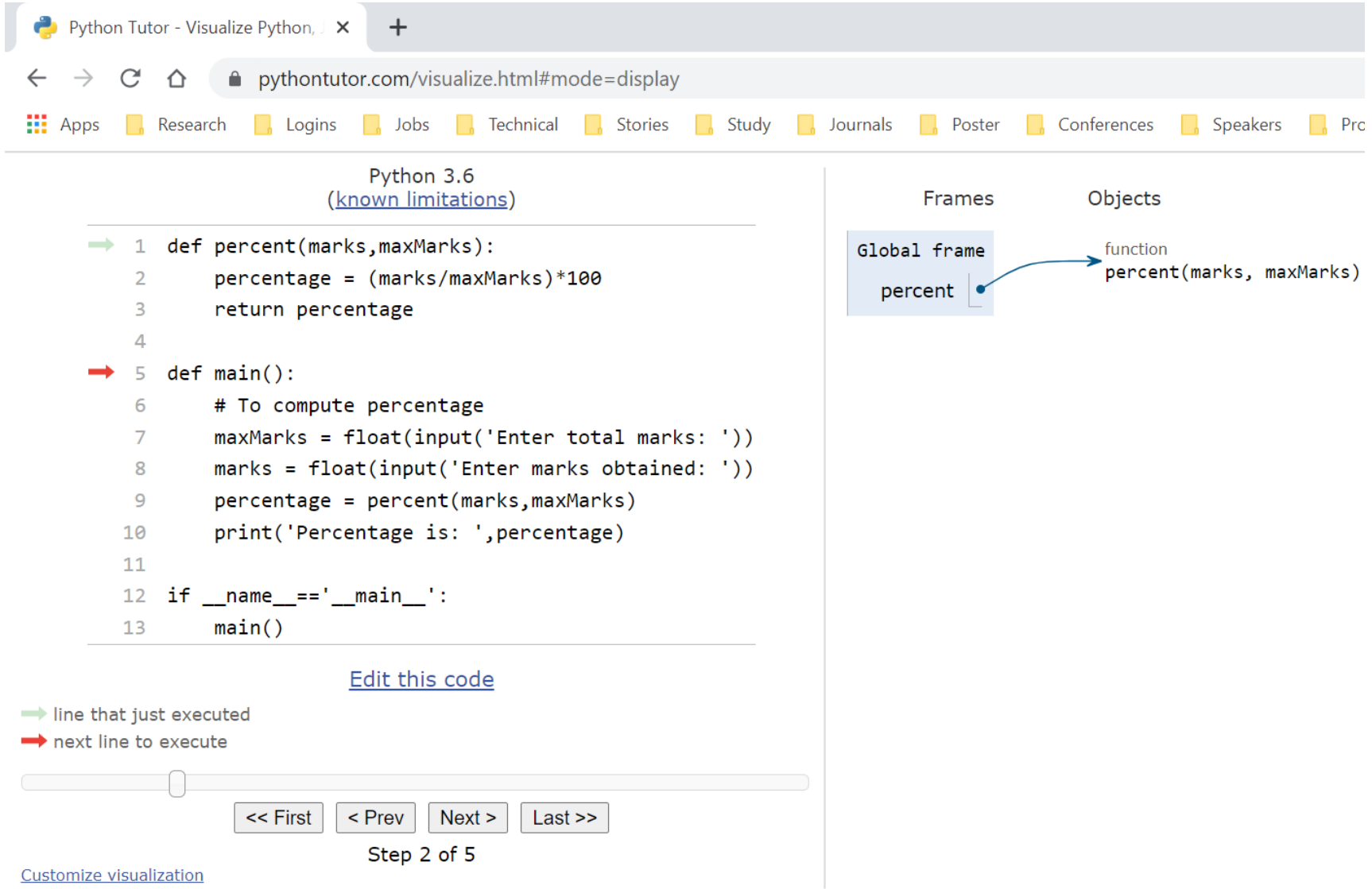
Visualization of Example-II (continued)

- On clicking **<Next>**, step 1 is executed and we see the function `percent` in the global frame



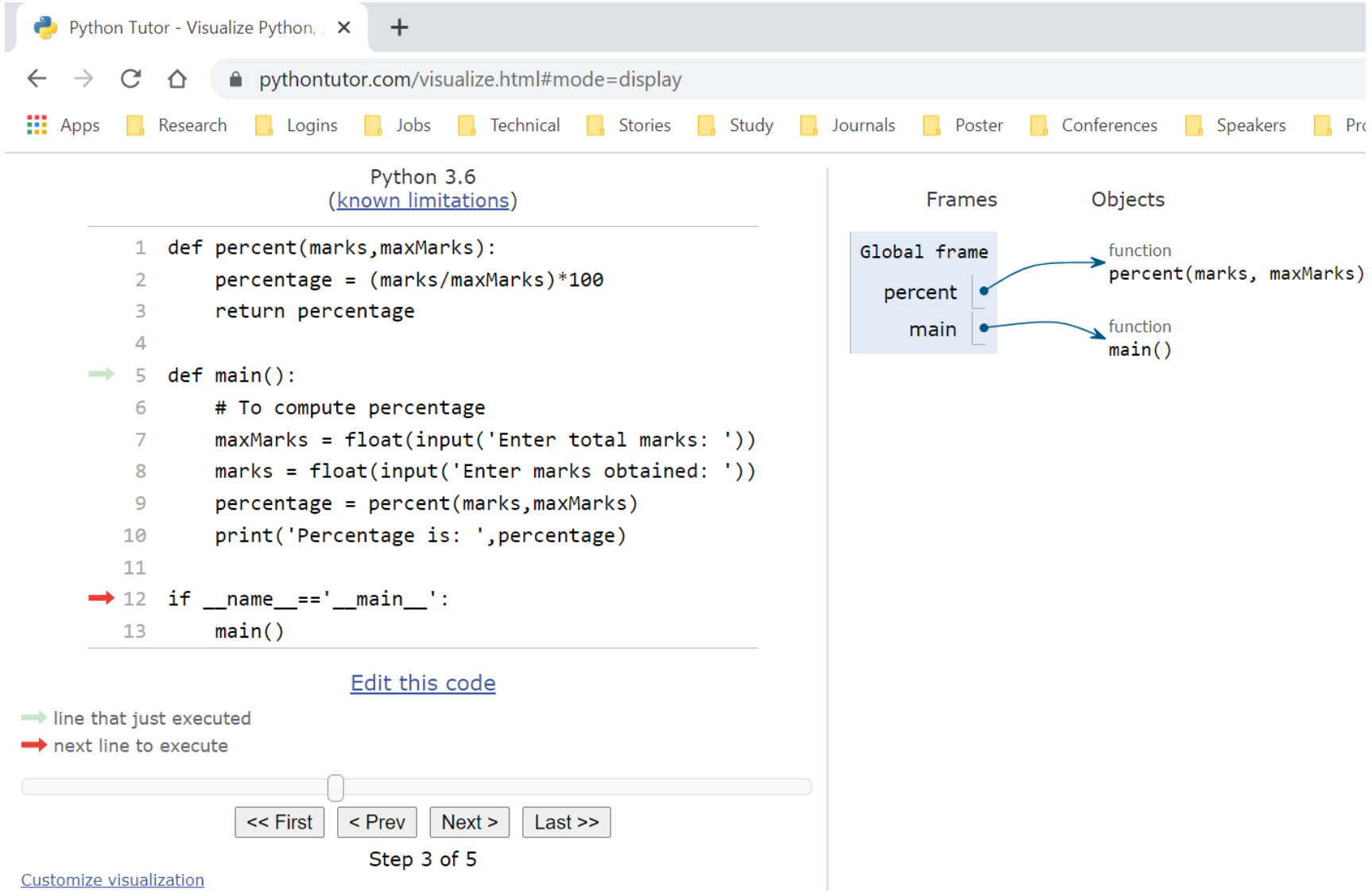
Visualization of Example-II (continued)

- On clicking **<Next>**, step 1 is executed and we see the function **`percent`** in the global frame
- On execution of next step, the function **`main`** is also shown in the global frame



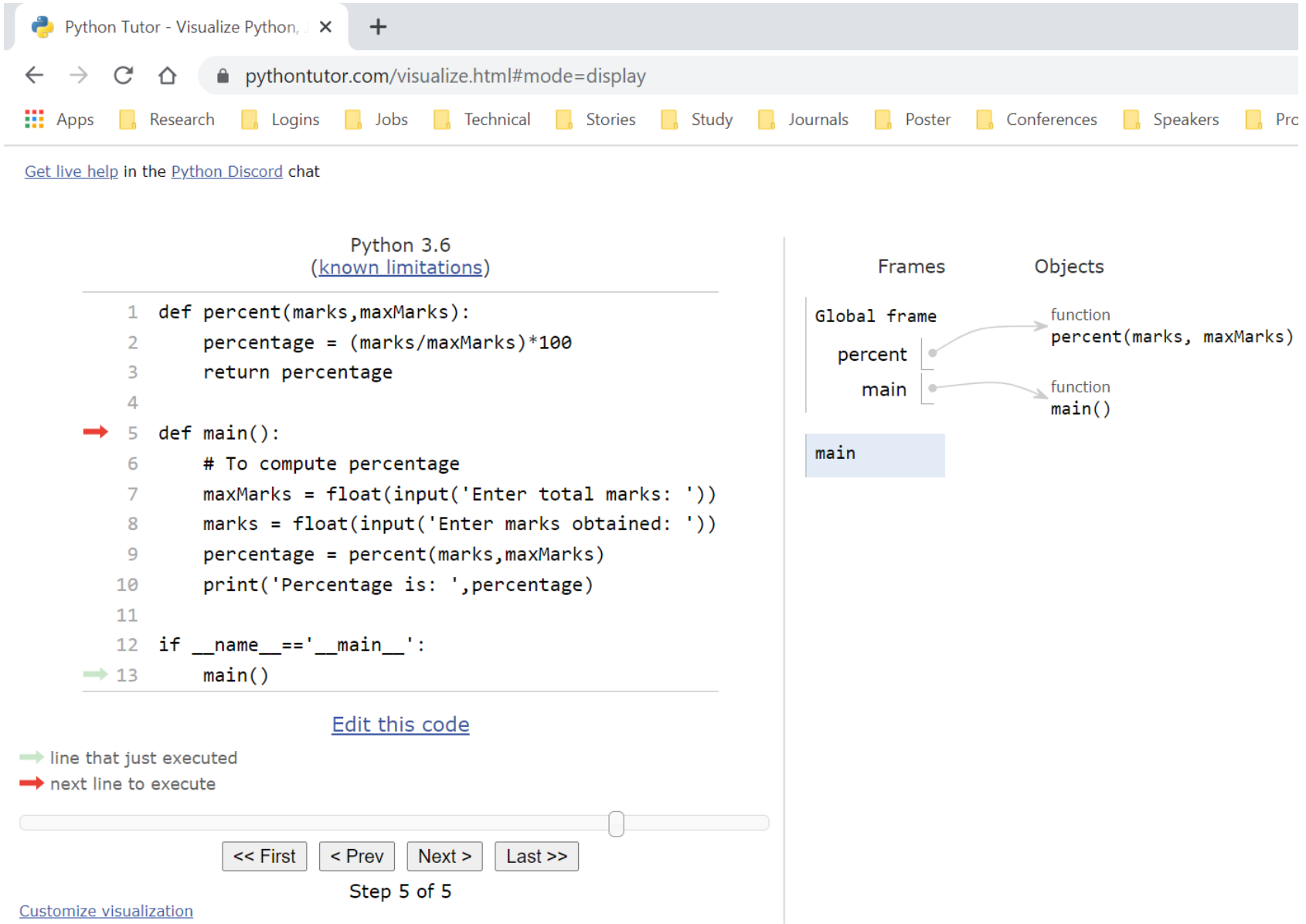
Visualization of Example-II (continued)

- When step 3 is executed, the condition `if __name__=='__main__'` evaluates as `True`
- Thus, in step 4 the function `main` is invoked



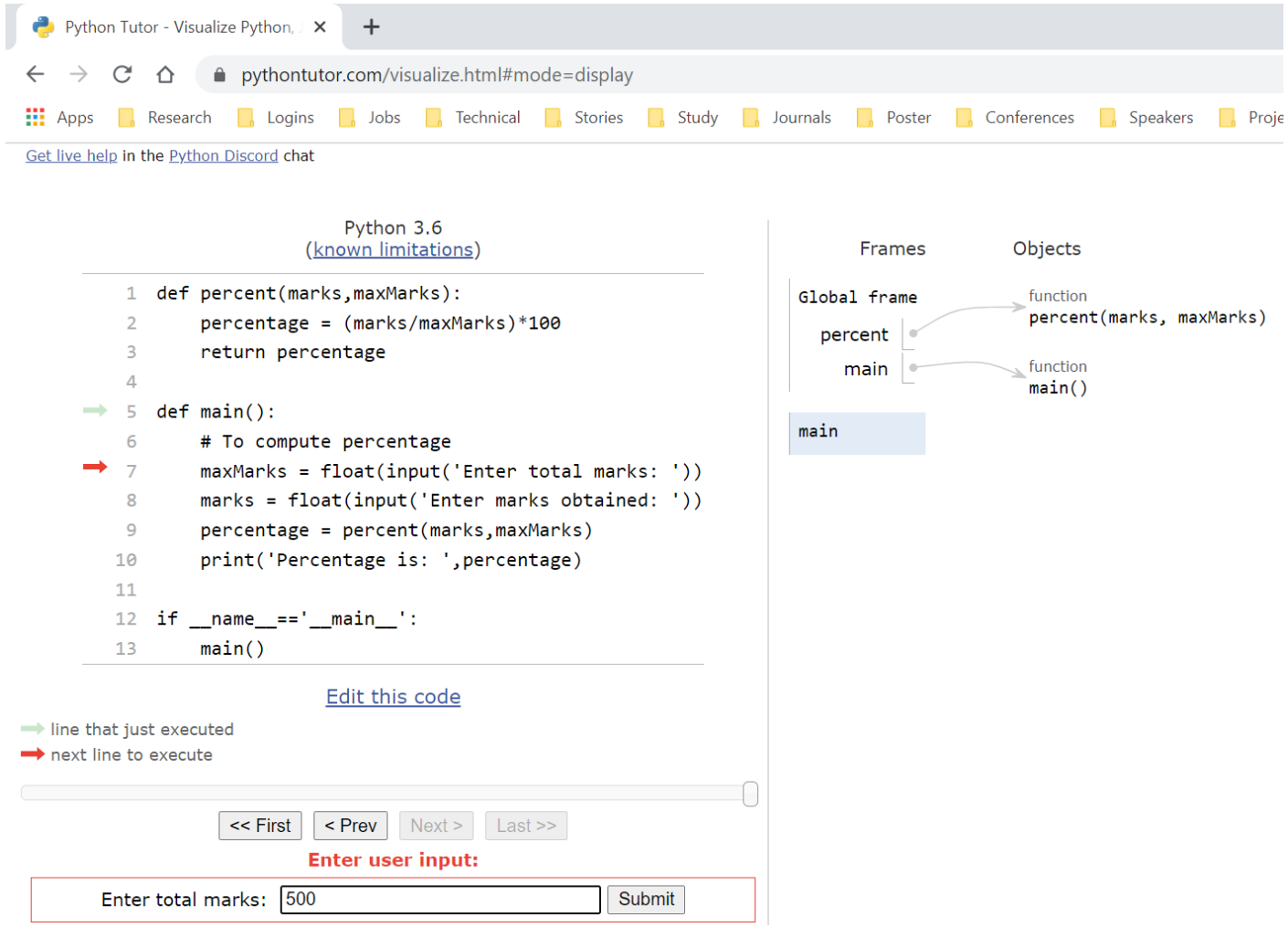
Visualization of Example-II (continued)

- Next click executes the call to function **`main`** and the visualizer shows the function **`main`** among frames



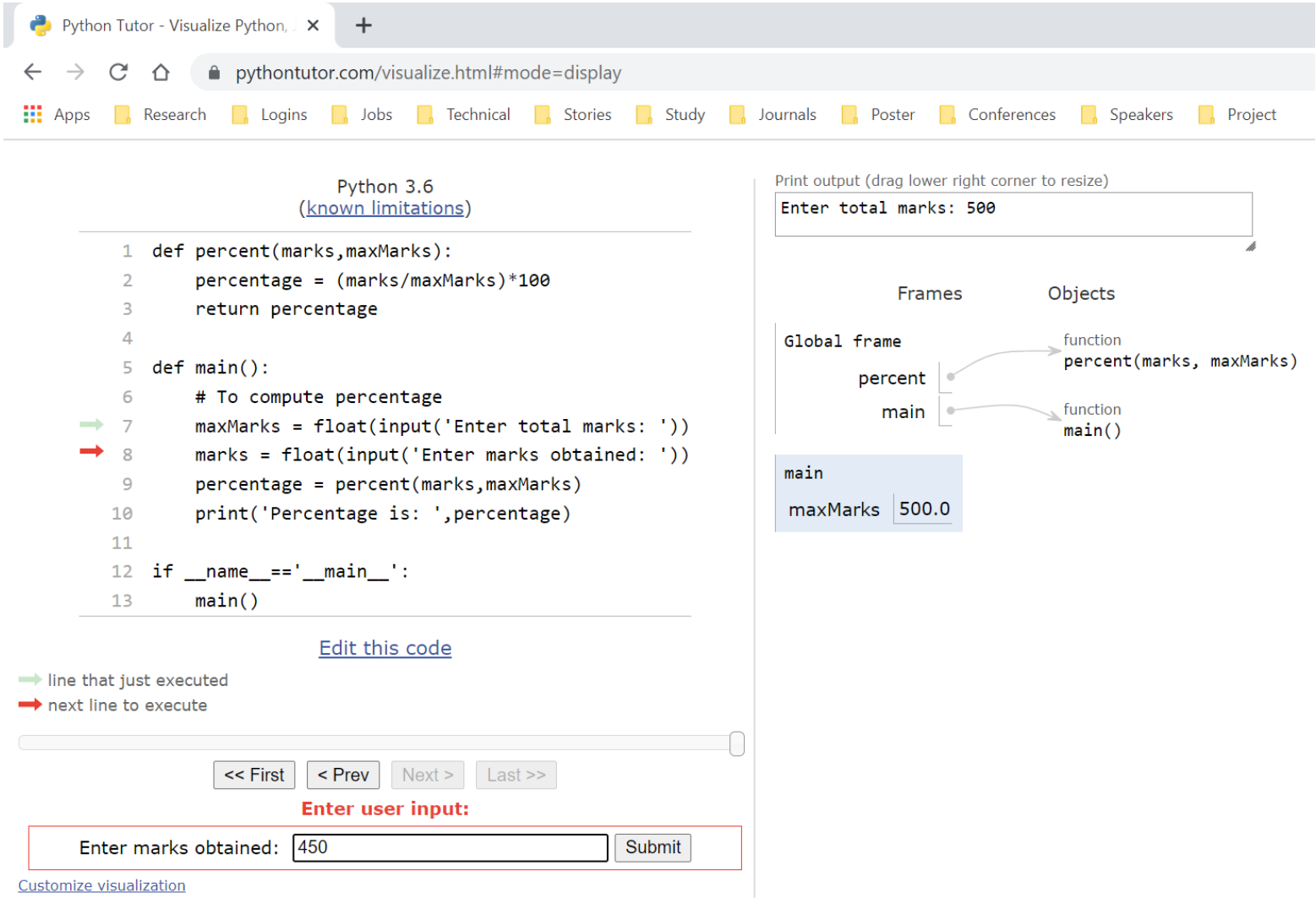
Visualization of Example-II (continued)

- Next click yields a message that prompts the user to enter total marks in **<Input Box>** and hit the **Submit** button
- This input message along with the values entered as input are also shown in **<Print output>**
- At this stage, we enter `500` as total marks, and click **Submit**



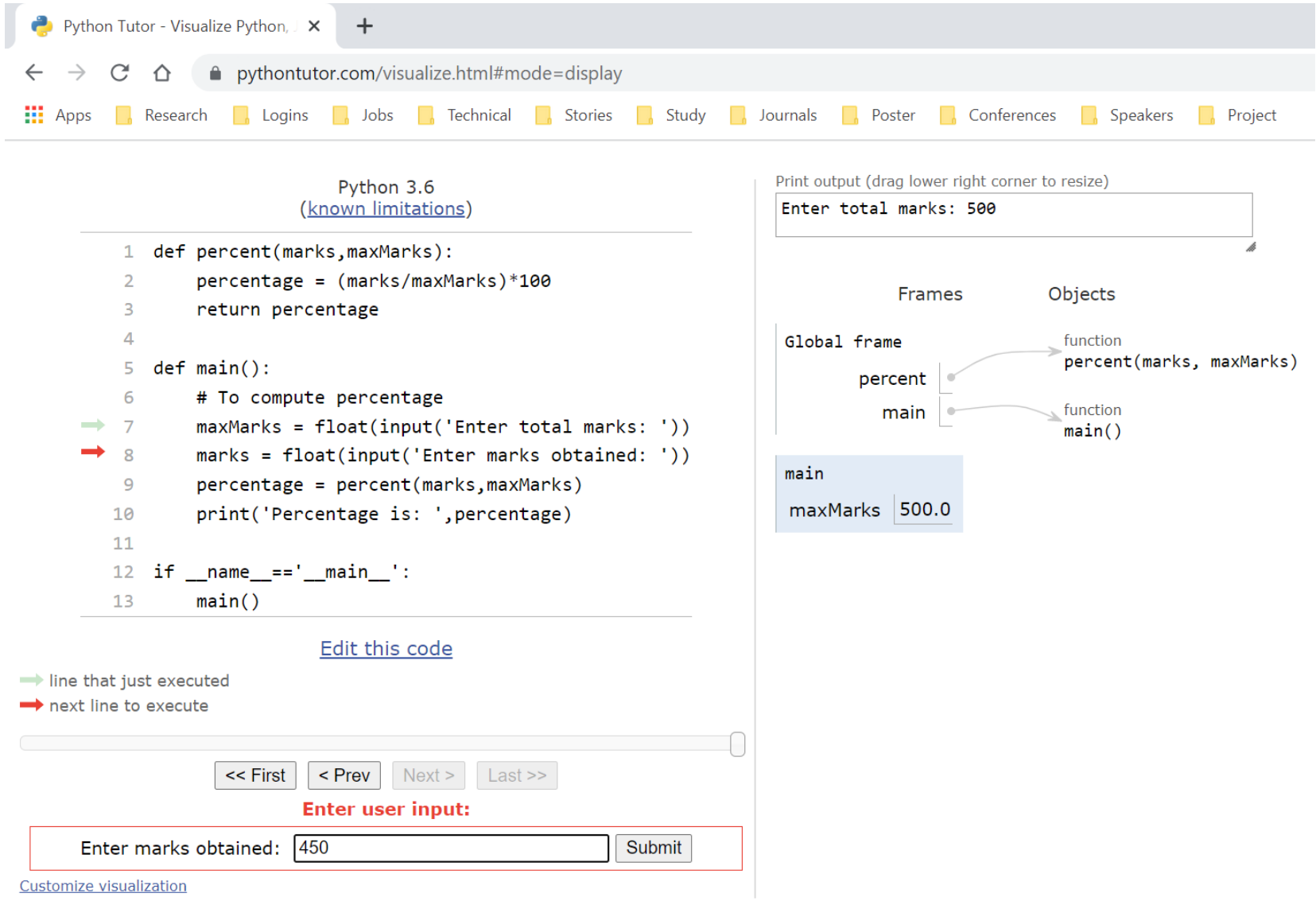
Visualization of Example-II (continued)

- Now the execution is resulting in creation of an instance of **`float`** object **`500.0`**
- This object is named as **`maxMarks`**
- The next click executes again prompting for marks obtained (say, **`450`**)



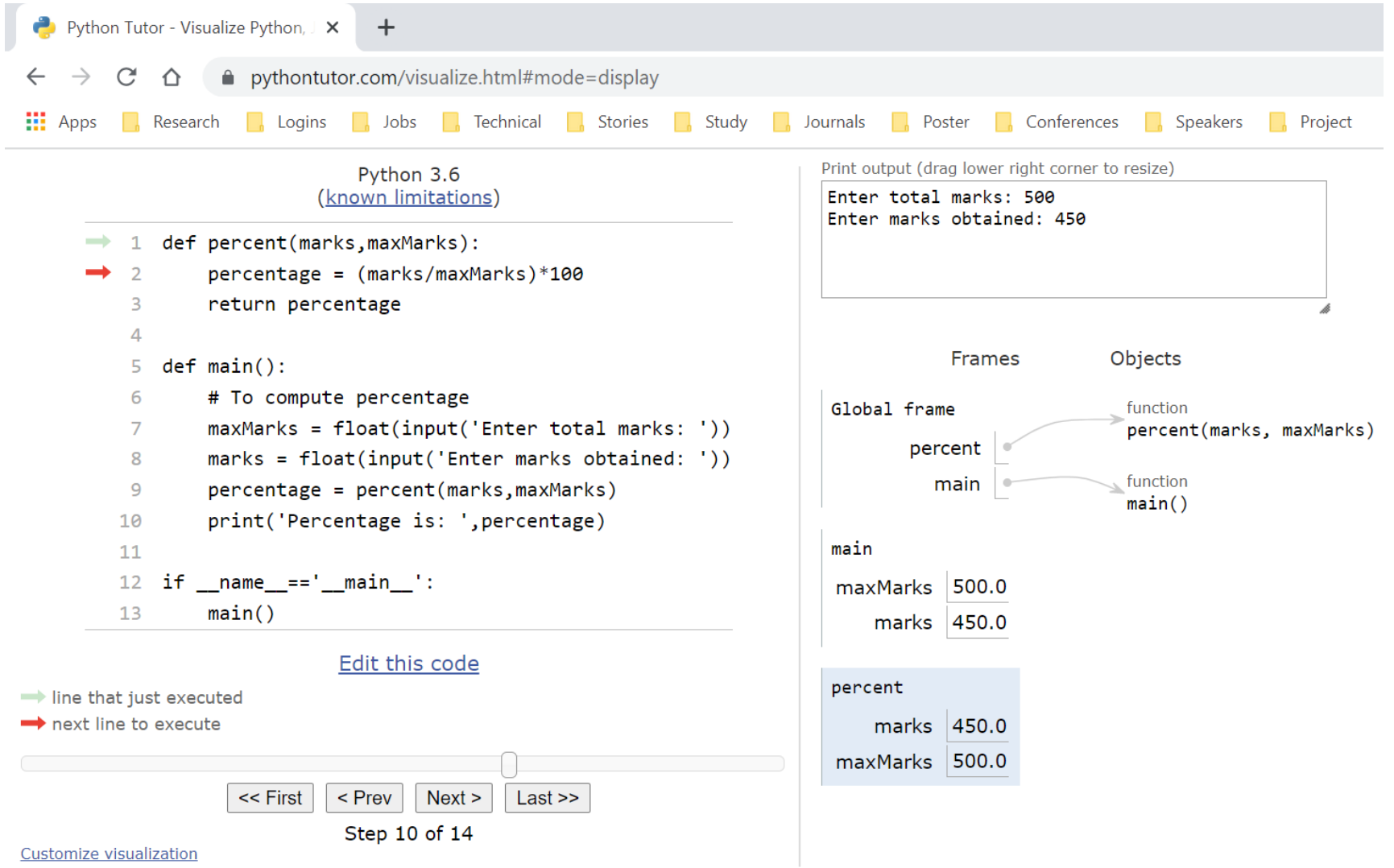
Visualization of Example-II (continued)

- As before, an instance of **`float`** object **`450.0`** is created, and named as `marks`
- The control moves the next line



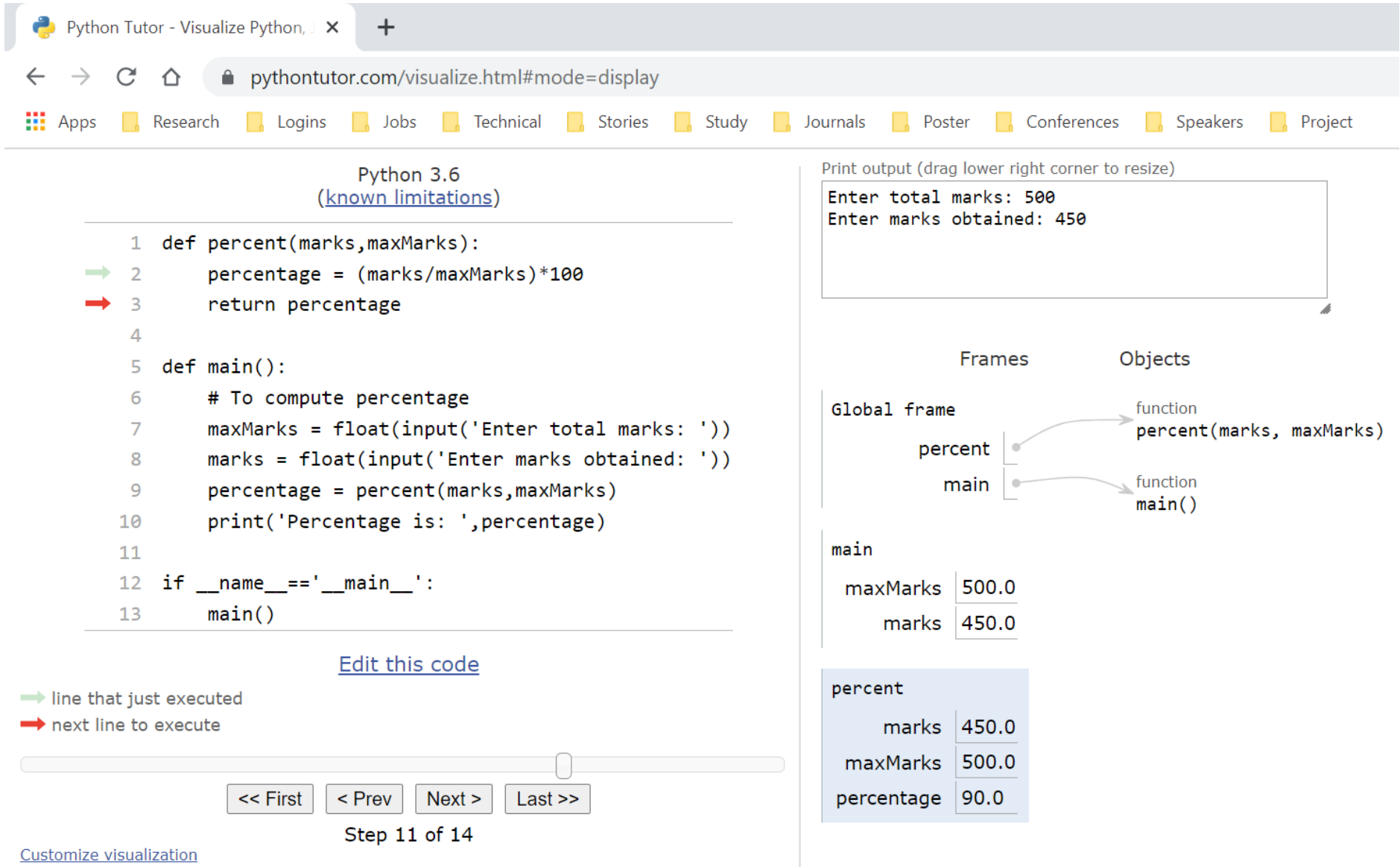
Visualization of Example-II (continued)

- Clicking **<Next>** executes the call to the function **`percent`** and the visualizer shows the function **`percent`** among frames
- Note that the parameters **`marks`** and **`maxMarks`** are mapped to **`float`** objects created earlier



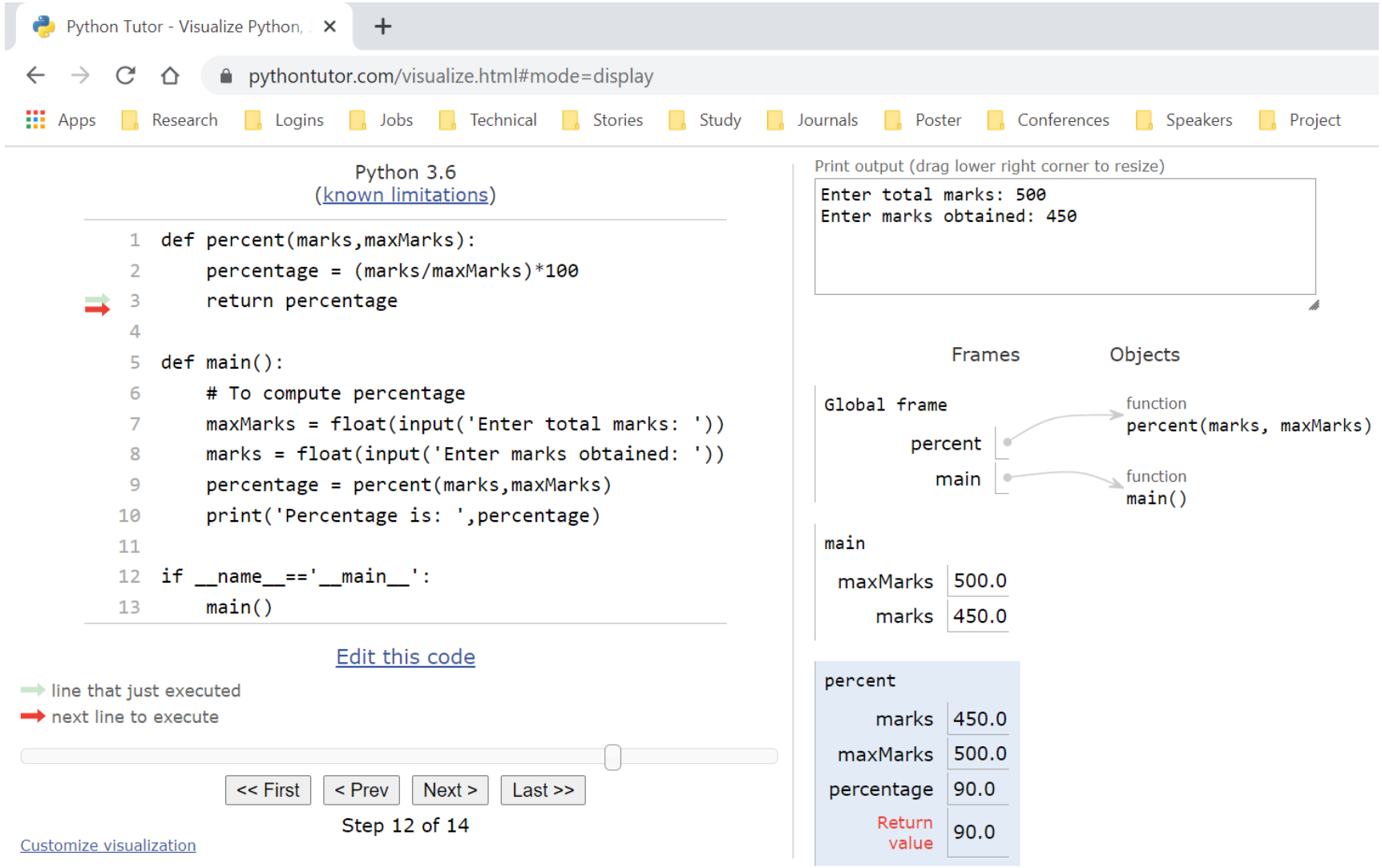
Visualization of Example-II (continued)

- On the next click, Python creates a return value i.e. the float object (to be returned to the **`main`** function) **`90.0`** which was created earlier



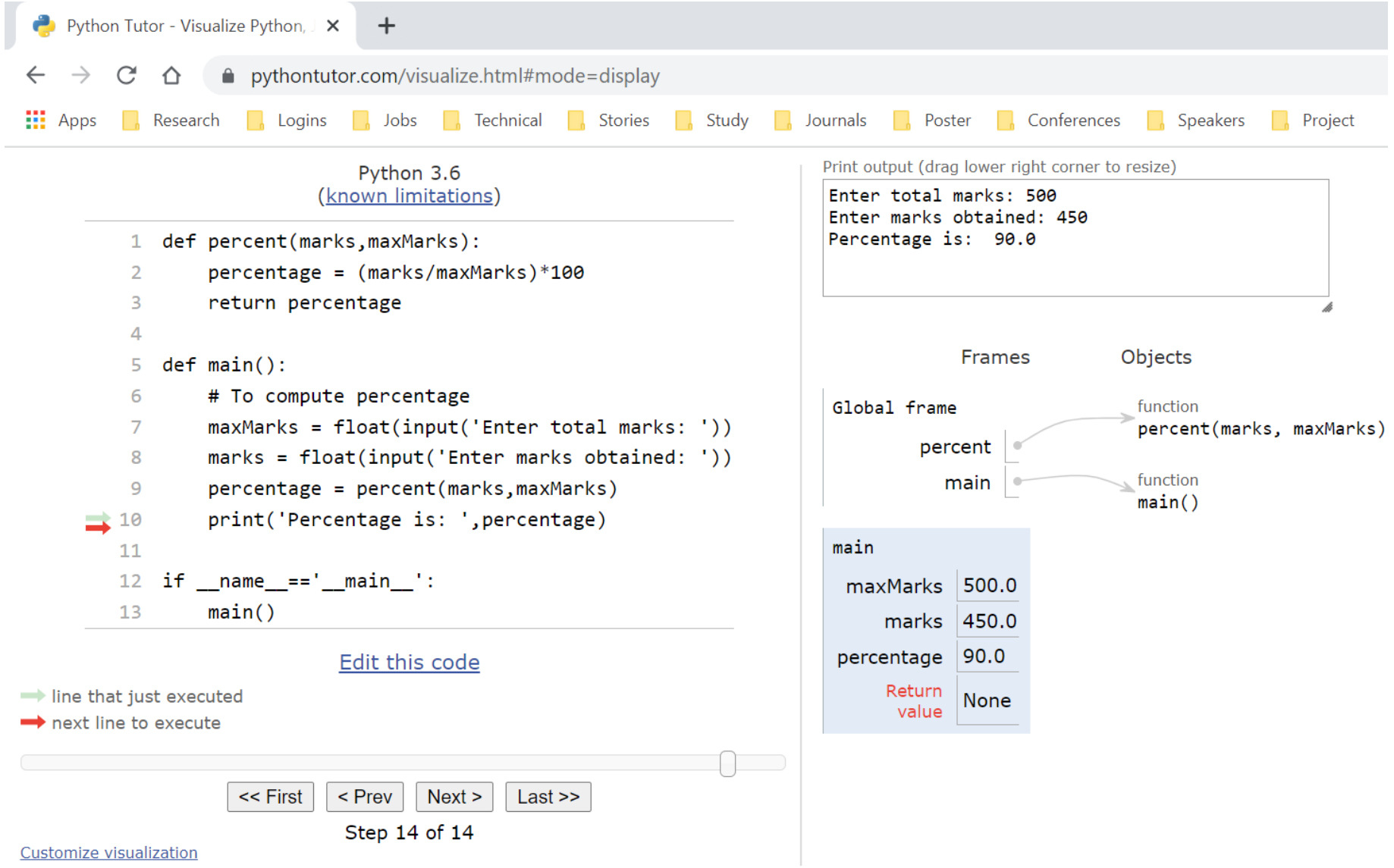
Visualization of Example-II (continued)

- The next click returns the value **`90.0`** from the function **`percent`** by associating it with the variable **`percentage`** of function **`main`**
- Note that the variable **`percentage`** of the function **`main`** now maps to the **`float`** object **`90.0`**



Visualization of Example-II

- The next two clicks shows return value **`None`** associated with the function **`main()`** as it does not return any value



Namespaces

- As the term suggests, a namespace is a space that holds some names
- A namespace defines a mapping of names to the associated objects
- In Python, a module, class, or function defines a namespace
- Names that appear in global frame (usually outside of the definition of classes, functions, and objects) are called global names and collectively they define the namespace called global namespace
- The names introduced in a class or function are said to be local to it.
- The region in a script in which a name is accessible is called its scope
- Thus, the scope of a name is resolved in the context of the namespace in which it is defined

*Python Programming: A Modular Approach by Sheetal Taneja and Kumar Naveen, Pearson Education India, Inc., 2017

An example

- For example, each of the functions **`f1`** and **`f2`** defined in a script may have the name **`x`**
- The variable **`x`** defined in function **`f1`** may refer to an object of type different from that of the object associated with variable **`x`** in the function **`f2`**
- A namespace maps names to objects
- Being an object-oriented language, definitions of functions and classes are also examples of objects

*Python Programming: A Modular Approach by Sheetal Taneja and Kumar Naveen, Pearson Education India, Inc., 2017

Scope

- The scope rules for names in Python are often summarized as LEGB rule
- LEGB stands for local, enclosing, global, and built in
- All names defined within the body of a function are local to it
- Function parameters are also considered local
- If a name is not locally found, Python recursively searches for its definition in an enclosing scope
- Names defined in the Python script but usually outside of any function or class definition are called global
- Python defines some built-in names such as `len` and `abs`, which can be accessed from anywhere in a program

*Python Programming: A Modular Approach by Sheetal Taneja and Kumar Naveen, Pearson Education India, Inc., 2017

Examples

- Example1: Note that as the variable **a** has global scope, it is accessible in function **f**

```
1   a = 4
2   def f():
3       print('global a: ', a)
4   f()
```

- Example 2: Note that the name **a** introduced in line 3 in the function **f** is local to it and has associated value **5**. Thus defining the value of name **a** in the function **f**, does not affect the value of the global name **a**.

```
1   a = 4.2
2   def f():
3       a = 5
4       print('local a: ', a)
5   f()
6   print('global a: ', a)
```

Examples (continued)

- Example 3: In this example, during execution of function **g**, when the variable **a** is to be accessed, Python looks for it in the local scope of function **g**, as it is not defined in the body of function **g**, it is searched in the next enclosing scope, i.e., the scope of function **f**, where the variable **a** is indeed defined, and therefore the value **5** of the variable **a** in function **f** gets bound to the occurrence of variable **a** in the function **g**.

```
1   a = 6
2   def f():
3       a = 5
4       def g():
5           b = a
6           print('inside function  g,  b: ', b)
7       g()
8   f()
```

- Example 4: In this example, the variable **a** is defined in the body of inner function **g**. When we attempt to access the name **a** in line 5 of the outer function **f**, Python looks for its definition first inside the body of function **f**, as there is no definition of **a** in the function **f**, it looks for definition of **a** in the next available enclosing scope, which is the global scope. Again, there is no definition of the name **a** in global name space, and hence the error message is printed.

```
1   def f():
2       def g():
3           a = 5
4       g()
5       print('in outer function g, a =', a)
6   f()
```

Conclusions

- Python visualizer is an online tool for visualizing the execution of Python code.
- A namespace defines a mapping of names to the associated objects.
- Names that appear in global frame outside of the definition of classes, functions, and objects are called global names, and collectively they define the namespace called global namespace.
- The names introduced in a class, or function are said to be local to it.
- The region in a script in which a name is accessible is called its scope.
- The scope rules for names in Python are often summarized as LEGB rule
- If a name is not locally defined, Python recursively searches for its definition in an enclosing scope.
- Python defines some built-in names such as `len` and `abs` which can be accessed from anywhere in a program.

Strings

Lecture 6

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- A string is a sequence of characters represented using single, double, or triple quotes.
  ⟩⟩⟩ message = 'Hello Gita'
- Triple quotes are typically used for strings that span multiple lines.
- *len* function finds the length of a string
  ⟩⟩⟩ len(message)
  10

Introduction (Cont.)

Individual characters within a string are accessed using a technique known as indexing.

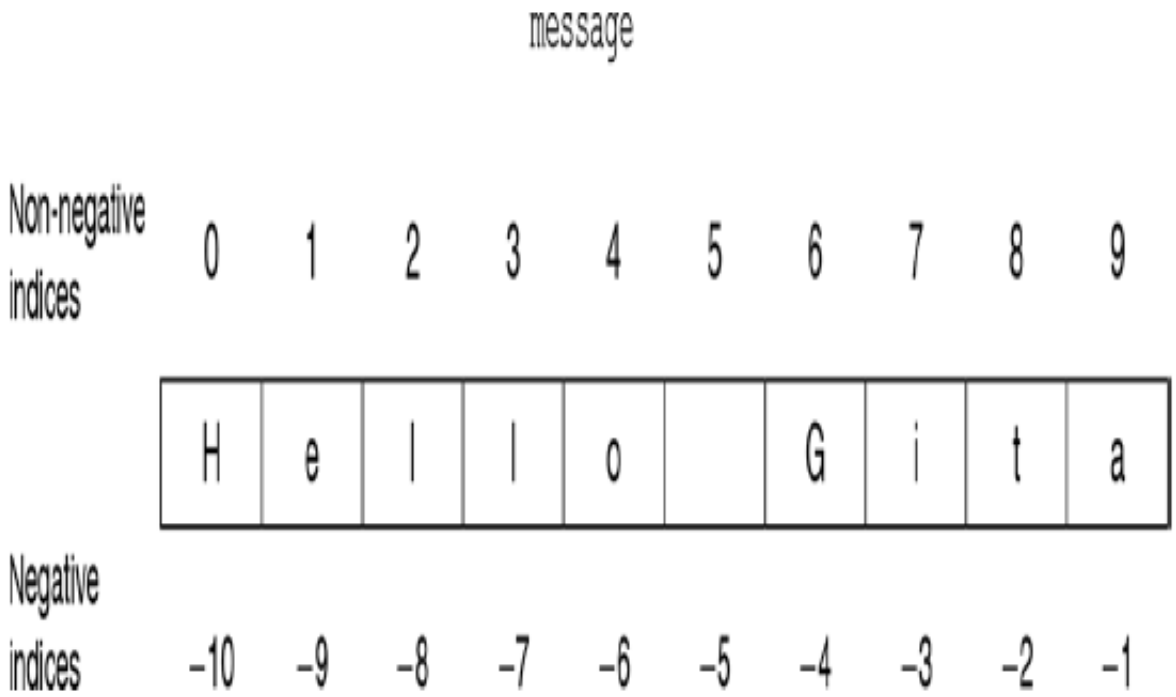


Figure 1: Indexing of variable **message**

Introduction (Cont.)

- An index is specified within the square brackets to access individual characters in a string via indices, for example:
  ⟩⟩⟩ message[6]
  'G'
  ⟩⟩⟩ index = len(message) - 1
  ⟩⟩⟩ message[index]
  'a'
- The negative indices range from - (length of the string) to -1.
- The entire range of valid indices would be {-10, -9, . . . , -1, 0, 1, . . . , 9}.
  ⟩⟩⟩ message[-1]
  'a'
  ⟩⟩⟩ message[-index]
  'e'

Introduction (Cont.)

```
⟩⟩⟩ message[15]
Traceback (most recent call last):
File "⟨pyshell#17⟩", line 1, in ⟨module⟩
message[15]
IndexError: string index out of range
```

Strings in Python are immutable.

```
⟩⟩⟩ message[6] = 'S'
Traceback (most recent call last):
File "⟨pyshell#2⟩", line 1, in ⟨module⟩
message[6] = 'S'
TypeError: 'str' object does not support item assignment
```

Introduction (Cont.)

```
⟩⟩⟩ 'Computer' + ' Science'
'Computer Science'
⟩⟩⟩ 'Hi'*3
'HiHiHi'
```

max() and min() are used to find maximum and minimum respectively of values.

```
⟩⟩⟩ max('AZ', 'C', 'BD', 'BT')
'C'
⟩⟩⟩ min('BD', 'AZ', 'C')
'AZ'
⟩⟩⟩ max('hello', 'How', 'Are', 'You', 'sir')
'sir'
```

Slicing

- To retrieve a sub-string, also called a slice, from a string.
  ⟩⟩⟩ message = 'Hello Sita'
  ⟩⟩⟩ message[0:5]
  'Hello'
  ⟩⟩⟩ message[-10:-5]
  'Hello'
- Python assumes 0 as the default value of start, and length of the string as the default value of end.
  ⟩⟩⟩ message[:5]
  'Hello'
  ⟩⟩⟩ message[5:]
  ' Sita'
  ⟩⟩⟩ message[:]
  "Hello Sita'

Slicing (Cont.)

```
⟩⟩⟩ message[:5] + message[5:]
'Hello Sita'
⟩⟩⟩ message[:15]
'Hello Sita'
⟩⟩⟩ message[15:]
''
⟩⟩⟩ message[:15] + message[15:]
'Hello Sita'
⟩⟩⟩ message[6:None]
'Sita'
```

Membership

- Python allows us to check for membership of the individual characters or sub-strings in strings using *in* operator.
- The expression s in str1 yields True or False depending on whether s is a sub-string of str1, for example:
  ⟩⟩⟩ 'h' in 'hello'
  True
  ⟩⟩⟩ 'ell' in 'hello'
  True
  ⟩⟩⟩ 'h' in 'Hello'
  False
- For loop can be used to iterate over each element in a sequence.

Built-in Functions on Strings (Cont.)

Functions	Explanation
s.count(str1)	counts number of times string str1 occurs in the string s
s.find(str1)	Returns index of the first occurence of the string str1 in string s, and returns -1 if str1 is not present in string s
s.rfind(str1)	Returns index of the last occurence of string str1 in string s, and returns -1 if str1 is not present in string s
s.capitalize(str1)	Returns a string that has first letter of the string s in uppercase and rest of the letters in lowercase

Built-in Functions on Strings (Cont.)

s.title()	returns a string that has first letter of every word in the string s in uppercase and rest of the letters in the lowercase
s.lower()	returns a string that has all uppercase letters in string s converted into corresponding lowercase letters
s.upper()	returns a string that has all lowercase letters in string s converted into corresponding uppercase letters
s.swapcase()	returns a string that has all lowercase letters in string s converted into uppercase letters and vice-versa

Built-in Functions on Strings (Cont.)

s.islower()	returns True if all alphabets in string s (comprising atleast one alphabet) are in lowercase, else returns False
s.isupper()	returns True if all alphabets in string s (comprising atleast one alphabet) are in uppercase, else returns False
s.istitle()	returns True if string s is in the title case, i.e., only first letter of each word is capitalized and the string s contains atleast one alphabet, and returns False otherwise
s.replace(str1, str2)	returns a string that has every occurence of string str1 in s replaced by string str2

Built-in Functions on Strings (Cont.)

s.strip()	returns a string that has whitespaces in s removed from the begining and the end
s.lstrip()	returns a string that has whitespaces in s removed from the begining
s.rstrip()	returns a string that has whitespaces in s removed from the end
s.split(delimiter)	returns a list formed by splitting the string s into substrings. The delimiter is used to mark the split points
s.partition (delimiter)	partitions the string s into three parts based on delimiter

Built-in Functions on Strings (Cont.)

s.join(sequence)	returns a string comprising elements of the sequence separated by delimiter s
s.isspace()	returns true if all characters in string s comprisewhitespace characters only, i.e., ' ','\n', and '\t' else False
s.isalpha()	returns true if all characters in string s comprise alphabets only, and False otherwise
s.isdigit()	returns true if all characters in string s comprisedigits only, and false otherwise
s.isalnum()	returns true if all characters in string s comprise alphabets and digits only, and false otherwise

Built-in Functions on Strings (Cont.)

s.startswith(str1)	returns true if string s starts with string str1, and false otherwise
s.endswith(str1)	returns true if string s ends with string str1, and false otherwise
s.encode(encoding)	returns s in an encoded form, based on the given encoding scheme
s.decode(encoding)	returns the decoded string s, based on the encoding scheme

Table 1: String Functions

Built-in Functions on Strings (Cont.)

```
⟩⟩⟩ 'Encyclopedia'.count('c')
2
⟩⟩⟩ colors = 'green, red, blue, red, red, green'
colors.find('red')
7
⟩⟩⟩ colors.rfind('red')
23
⟩⟩⟩ colors.find('orange')
-1
⟩⟩⟩ 'python IS a Language'.capitalize()
'Python is a language'
⟩⟩⟩ 'python IS a PROGRAMMING Language'.title()
'Python Is A Programming Language'
```

Built-in Functions on Strings (Cont.)

```
⟩⟩⟩ emailId1 = 'geek@gmail.com'
⟩⟩⟩ emailId2 = 'Geek@gmail.com'
⟩⟩⟩ emailId1 == emailId2
False
⟩⟩⟩ emailId1.lower() == emailId2.lower()
True
⟩⟩⟩ emailId1.upper() == emailId2.upper()
True
⟩⟩⟩ 'pYTHON IS PROGRAMMING LANGUAGE'.swapcase()
'Python is programming language'
⟩⟩⟩ 'python'.islower()
True
⟩⟩⟩ 'Python".isupper()
False
```

Built-in Functions on Strings (Cont.)

```
⟩⟩⟩ 'Basic Python Programming'.istitle()
True
⟩⟩⟩ 'Basic PYTHON Programming'.istitle()
False
⟩⟩⟩ '123'.istitle()
False
⟩⟩⟩ 'Book 123'.istitle()
True
⟩⟩⟩ message = 'Amey my friend, Amey my guide'
⟩⟩⟩ message.replace('Amey', 'Vihan')
'Vihan my friend, Vihan my guide'
```

Built-in Functions on Strings (Cont.)

```
⟩⟩⟩ ' Hello How are you! '.lstrip()
'Hello How are you! '
⟩⟩⟩ ' Hello How are you! '.rstrip()
' Hello How are you!'
⟩⟩⟩ ' Hello How are you! '.strip()
'Hello How are you!'
⟩⟩⟩ colors = 'Red, Green, Blue, Orange, Yellow, Cyan'
⟩⟩⟩ colors.split(',')
['Red', ' Green', ' Blue', ' Orange', ' Yellow', ' Cyan']
⟩⟩⟩ colors.split()
['Red,', 'Green,', 'Blue,', 'Orange,', 'Yellow,', 'Cyan']
⟩⟩⟩ 'Hello. How are you?'.partition('.')
('Hello', '.', ' How are you?')
```

Built-in Functions on Strings (Cont.)

```
>>> ' '.join(['I',' am',' ok'])
'I > am > ok'
>>> ' '.join(('I', 'am', 'ok'))
'I am ok'
>>> ' '.join("'I',' am',' ok'" )
"'I', ', am', ', ok'"
>>> name = input('Enter your name : ')
>>> Enter your name : Nikhil
>>> name.isalpha()
True
>>> name = input('Enter your name : ')
>>> Enter your name : Nikhil Kumar
>>> name.isalpha()
False
```

Built-in Functions on Strings (Cont.)

```
>>> mobileN = input('Enter mobile no : ')
Enter mobile no : 1234567890
>>> mobileN.isdigit() and len(mobileN) == 10
True
>>> ' '.isspace()
True
>>> password = input('Enter password : ')
Enter password : Kailash107Ganga
>>> password.isalnum()
True
>>> password = input('Enter password : ')
Enter password : Kailash 107 Ganga
>>> password.isalnum()
False
>>> name = 'Ankita Narain Talwar'
>>> name.endswith('Talwar')
True
```

Built-in Functions on Strings (Cont.)

```
⟩⟩⟩ name = 'Dr. Vihan Garg'
⟩⟩⟩ name.startswith('Dr. ')
True
⟩⟩⟩ name = ' Dr. Amey Gupta'
⟩⟩⟩ name.startswith('Dr. ')
False
⟩⟩⟩ str1 = 'message'.encode('utf32')
⟩⟩⟩ str1
b'\xff\xfe\x00\x00m\x00\x00\x00e\x00\x00\x00s\x00\x00\x00s\
⟩⟩⟩ str1.decode('utf32')
'message'
```

Note that the original string S remains unchanged in each case.

Counting the Number of Matching Characters in a Pair of Strings

```
def nMatchedChar(str1, str2):
    '''
    Objective: to count number of occurrences of characters in
str1 that are also in str2
    Input parameters: str1,str2-string
    Return value: count-numeric
    '''
    temp1 = str1.lower()
    temp2 = str2.lower()
    count=0
    for ch1 in temp1:
        #search for ch1 in temp2
            for ch2 in temp2:
                if ch1==ch2:
                    count+=1
    return count
```

Counting the Number of Matching Characters in a Pair of Strings (Cont.)

```
⟩⟩⟩ name1 = 'Ram Rahim'
⟩⟩⟩ name2 = 'SAMARTH RAHI'
⟩⟩⟩ nMatchedChar(name1, name2)
16
```

Counting the Number of Common Characters in a Pair of Strings

```
    '''
    Objective: to count number of occurrences of characters in
two strings
    Input parameters: str1,str2-string
    Return value: count-numeric
    '''
```

Counting the Number of Common Characters in a Pair of Strings

```
def nCommonChar(str1, str2):
        temp1 = str1.lower()
        temp2 = str2.lower()
        count=0
        for i in range(len(temp1)):
                ch1=temp1[i]
                if not(ch1 in temp1[:i]):
                #if the character has not been encountered earlier
                        for ch2 in temp2:
                                if ch1==ch2:
                                        count+=1
                                        break
        return count
```

Reversing a String

```
def reverse(str1):
        '''
        Objective: to reverse a string
        Input parameters: str1-string
        Return value: reverseStr-reverse of str1- string
        '''
        reverseStr = ''
        for i in range(len(str1)):
                reverseStr=str1[i] + reverseStr
        return reverseStr
```

Reversing a String (Cont.)

using recursion

```
def reverse(str1):
    '''
    Objective: to reverse a string
    Input parameters: str1-string
    Return value: reverse of str1- string
    '''
    if str1 == ''
        return str1
        else:
        return reverse(str1[1:]) + str1[0]
```

Pattern Matching

- **Alphabet (Σ)**: An alphabet is a non-empty set of symbols
- **String:** A string is a finite sequence of symbols chosen from the alphabet. An empty string ” containing no symbols is called null string and is often denoted by λ or ϵ. The length of a string is defined as the number of symbols in the string. The length of the null string is defined to be zero.
- **Language:** It is the set of strings (words) defined over the alphabet that conforms to some predefined pattern, or rule(s).

- A regular language is described by a regular expression
- regular expressions: used to define regular languages

Pattern Matching (Cont.)

- Each symbol of the alphabet defines a regular language, comprising the symbol itself.
- If r is regular expression, L(r) denotes the language described by r.
- λ or ϵ is a regular expression that denotes the language comprising the null string only.
- The language containing no word, not even λ is called null or empty language $\{\}$, and is denoted by the regular expression ϕ.
- If r and s are regular expressions, r|s is also a regular expression and denotes the language L(r|s) = L(r) U L(s).
- If r is a regular expression, so is (r), denoting the same language, i.e. L(r) = L((r)). Parentheses are used to enforce precedence of operators in the regular expressions.
- The regular expression rs denotes the language L(rs) = L(r) L(s)
- Concatenation has higher precedence than the union operator

Pattern Matching (Cont.)

- Concatenation is not commutative
- The regular expression (0|1)1 denotes the language L((0|1)1) = L((0|1))L(1) = {0,1} {1} = {01,11}.
- If r is a regular expression, r* is also a regular expression. * denotes **zero or more** occurrences of the preceding pattern r.
- 0* defined over alphabet $\Sigma = \{0, 1\}$ defines the language, L = $\{\lambda$, 0,00, 000, 0000, . . . $\}$
- Given a pattern r,

$$r^+$$

  denotes the language comprising **one or more** occurrences of strings that match the pattern r.
- For dealing with a regular expression in Python, we need to import the module re, which contains functions for handling the regular expressions

Pattern Matching (Cont.)

Symbols used in regular expression	**Meaning**
*	zero or more occurrences of the preceding pattern
.	exactly one arbitrary character excluding newline
?	zero or one occurrence of the preceding pattern
+	one or more occurrences of the preceding pattern
{m}	exactly m occurrences of the preceding pattern

Pattern Matching (Cont.)

{m,n}	at least m and at most n occurrences of preceding term. In absence of n, there is no upper bound and in the absence of m, the lower bound is assumed to be zero
{list-of-char}	a single character from list of characters enclosed between []
[.]	matches dot (not an arbitrary character)
[a-z]	a single character in the range a to z
[A-Z]	a single character in the range A to Z

Pattern Matching (Cont.)

[0-9]	A single digit in the range 0 to 9
[^...]	when ^ occurs at the begining of a list symbols enclosed between [], it denotes a single character not in the list
^	matches beginning of the string
$	matches end of the string or just before the newline character at the end of the string
r1|r2	regular expression r1 or r2

Pattern Matching (Cont.)

()	Groups pattern elements
\d	any digit
\D	any non-digit character
\s	whitespace character
\S	Non-whitespace character
\w	any alphanumeric character including _
\W	any non-alphanumeric character excluding _

Table 2: symbols used in regular expressions

Pattern Matching (Cont.)

regular expression	**set of matched patterns**
python	python
{p|P}ython	{python, Python}
a*	$\{\lambda, a, aa, aaa, ...\}$
a+	{, a, aa, aaa,...}
a?	$\{\lambda, a\}$
[aeiou]	{a,e,i,o,u}
[ab]?	$\{\lambda, a, b\}$
[ab]*	$\{\lambda, a, b, aa, ab, bb, aa, aab, ...\}$

Pattern Matching (Cont.)

regular expression	**set of matched patterns**
\d	{0,1,2,3,4,5,6,7,8,9}
\d{2}	{00,01,02,03,...,99}
\d{2,3}	{00,01,02,...99,000,001,002,....,999}
\D	{a,b,...,z,A,B,...,Z,, ∗, $, !, ...}
\w	{a,b,...,z,A,B,...,Z,0,1,2,...,9}
\s	{space, tab, newline, carriage return}
[^a-b]	the set comprising characters other than 0 and 1
(a|b)(c)$	{ac,bc}, there should be no character after c in the string that matches the pattern

Pattern Matching (Cont.)

^(a)(0|1)*	{a,a0,a1,a00,a01,..}, a should be first in the string which pattern matches
a*b	{a*b}, when the backslash character precedes a character with a special meaning, the special meaning of the character is ignored. Example, although the regular expression a*b defines a pattern comprising zero or more occurrences of a, followed by b, the pattern a*b defines the string 'a*b'

Table 3: Examples of regular expressions and the corresponding languages

Pattern Matching (Cont.)

- The function **search** of this module is used for matching a regular expression in the given string.
- It looks for the first location of a match in the given string.
- If the search is successful, it returns the **matchObject** instance matching the regular expression pattern, otherwise it returns **None**.
- The function **group** of matchObject instance returns the substring that matches the regular expression.

Pattern Matching (Cont.)

Regular Expression	**Example**
Python	⟩⟩⟩ string1 = 'Welcome to python shell' ⟩⟩⟩ match = re.search('python', string1) ⟩⟩⟩ match.group() 'python'
{p|P}ython	⟩⟩⟩ string1 = 'Welcome to Python shell' ⟩⟩⟩ match = re.search('{p|P}ython', string1) ⟩⟩⟩ match.group() 'Python'
Shel*	⟩⟩⟩ string1 = 'Python shell' ⟩⟩⟩ match = re.search('Shel*', string1) ⟩⟩⟩ match.group() 'Shell'

Pattern Matching (Cont.)

Shel?	⟩⟩⟩ string1 = 'Python shell' ⟩⟩⟩ match = re.search('Shel?', string1) ⟩⟩⟩ match.group() 'Shel'
Shel{1,2}	⟩⟩⟩ string1 = 'Python shellll' ⟩⟩⟩ match = re.search('Shel{1,2}', string1) ⟩⟩⟩ match.group() 'Shell'
.....	⟩⟩⟩ string1 = 'Python shell' ⟩⟩⟩ if re.search('.....', string1): print('String length is greater than or equal to 5') String length is greater than or equal to 5

Pattern Matching (Cont.)

^Python	>>> string1 = 'Python is a powerful language'
	>>> if(re.search('^Python', string1))
	print('String starts with python')
	'String starts with python'

^power	>>> string1 = 'Python is a powerful language'
	>>> if(re.search('^power', string1))
	print('String starts with power')
	else:
	print('String does not start with power')
	String does not start with power

powerful$	>>> string1 = 'Python is a powerful language'
	>>> if(re.search('powerful$', string1))
	print('String ends with powerful')
	else:
	print('String does not ends with powerful')
	String does not ends with powerful

Pattern Matching (Cont.)

language$	⟩⟩⟩ string1 = 'Python is a powerful language' ⟩⟩⟩ if(re.search('language$', string1)) print('String ends with language') else: print('String does not ends with language') String ends with language
\d\d\d\d\d	⟩⟩⟩ string1 = 'Roll number is 23456' ⟩⟩⟩ match=re.search('\d\d\d\d\d', string1) ⟩⟩⟩ match.group() '23456'
\d{5}	⟩⟩⟩ string1 = 'Roll number is 23456' ⟩⟩⟩ match=re.search('\d{5}', string1) ⟩⟩⟩ match.group() '23456'

Pattern Matching (Cont.)

-[0-9]+\.[0-9]+	⟩⟩⟩ string1 = 'Decrease in price is -45.89' ⟩⟩⟩ match = re.search('-[0-9]+\.[0-9]+', string1)) match.group() -45.89
\w*	⟩⟩⟩ string1 = 'Python Shell' ⟩⟩⟩ match = re.search('\w*', string1)) match.group() 'Python'
\w*\s\w*	⟩⟩⟩ string1 = 'We used Python Shell' ⟩⟩⟩ match = re.search('\w*\s\w*', string1)) match.group() 'We used'
.*	⟩⟩⟩ string1 = 'I use **Python**, do you?' ⟩⟩⟩ match = re.search('.*', string1)) match.group() 'I use **Python**, do you?'

Pattern Matching (Cont.)

(a(b|c))*	⟩⟩⟩ string1 = 'abac12ccaab' ⟩⟩⟩ match = re.search('(a(b|c))*', string1)) match.group() 'abac'
(a(b|c))*\d{1,2}c*	⟩⟩⟩ string1 = 'abac12ccaab' ⟩⟩⟩ match = re.search('(a(b|c))*\d{1,2}c*', string1)) match.group() 'abac12cc'
\w*\d\d.*b$	⟩⟩⟩ string1 = 'abac12ccaab' ⟩⟩⟩ match = re.search('\w*\d\d.*b$\s\w*', string1)) match.group() 'abac12ccaab'

Pattern Matching (Cont.)

(a(b|c))*	⟩⟩⟩ string1 = 'abac12ccaab' ⟩⟩⟩ match = re.search('(a(b|c))*', string1)) match.group() 'abac'
(a(b|c))+	⟩⟩⟩ string1 = 'abac12ccaab' ⟩⟩⟩ match = re.search('(a(b|c))+', string1)) match.group() 'abac'

Table 4: Python examples of regular expressions

Pattern Matching (Cont.)

To find email ids (pranav.gupta@cs.iitd.ac.in) from a string:

a sequence of alphanumeric characters	denoted by [a-z0-9]+
a repeating(0 or more times) sequence of dots followed by alphanumeric characters	denoted by (\.[a-z0-9]+)*
@	denoted by @
sequence of alphabetic characters	denoted by [a-z]+
repeating (1 or more times) sequence of dot followed by alphabetic characters	denoted by (\.[a-z]+)+

An email id may be represented using the regular expression [a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+.

Pattern Matching (Cont.)

- Generally, a regular expression is preceded with r to denote a raw string.
- Use of a raw string as a regular expression avoids any confusion with some characters that have special meaning in regular expressions.

⟩⟩⟩ match = re.search(r'[a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+', 'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com, raman@gmail.com')

⟩⟩⟩ match.group()

'ram@gmail'

Pattern Matching (Cont.)

- **finditer:** retrieving all substrings matching a regular expression.

```
⟩⟩⟩ for i in re.finditer(r'[a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+',
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,
raman@gmail.com'):
print(i.group())

'ram@gmail.com'
'pranav.gupta@cs.iitd.ac.in'
'nik@yahoo.com'
'raman@gmail.com'
```

Pattern Matching (Cont.)

- searching for words ending with **ing** in a string

```
⟩⟩⟩ for i in re.finditer(r'[A-Za-z]+ing', 'Walking down the road, he was thinking about the coming years.'):
⟩⟩⟩ print(i.group())
Walking
thinking
coming
```

- **findall:** retrieve a list of all the substrings matching a regular expression

```
⟩⟩⟩ for i in re.findall(r'[A-Za-z]+ing', 'Walking down the road, he was thinking about the coming years.'):
['Walking', 'thinking', 'coming']
```

Pattern Matching (Cont.)

```
⟩⟩⟩ message = 'Python:Python is an interactive language. It is
developed by Guido Van Rossum'
⟩⟩⟩ words = re.findall('\w+', message)
⟩⟩⟩ len(words)
13
⟩⟩⟩ re.findall(r'([a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+)',
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,
raman@gmail.com')
[('ram@gmail.com', '', '.com'), ('pranav.gupta@cs.iitd.ac.in', '.gupta',
'.in'), ('nik@yahoo.com', '', '.com'), ('raman@gmail.com', '', '.com')]
```

Pattern Matching (Cont.)

- **r**egular expression to extract all the single line comments

#.*

Example:

⟩⟩⟩ pythonCode = ”’
”””
Python code to add two numbers.
”””
a = 5 #number1
b = 5 #number2
Compute addition of two numbers
c = a + b
”’

⟩⟩⟩ for i in re.finditer(r’(#.*)’, pythonCode):
⟩⟩⟩ print(i.group())
#number1
#number2
#Compute addition of two numbers

Pattern Matching (Cont.)

- **r**egular expression to extract all the multi-line comments

”””.*?”””

Note: a dot in a regular expression includes any character except end of line. However, if we include re.DOTALL as the third argument in the function **finditer**, dot matches newline character also. For example:

```
⟩⟩⟩ for i in re.finditer(r’(”””.*?”””)’, pythonCode, re.DOTALL):
⟩⟩⟩ print(i.group())
```

”””

Python code to add two numbers.

”””

Pattern Matching (Cont.)

⟩⟩⟩ re.split(r',', 'Mira, Ronit, Vivek')
['Mira', 'Ronit', 'Vivek']

re.split (): returns a list of the substrings delimited by the regular expression provided
⟩⟩⟩ re.split(r',|\n', '''Mira,Rohit,Vivek
Aiysha,Renuka,Robin
Sneha,Ravi''')
['Mira', 'Rohit', 'Vivek', 'Aiysha', 'Renuka', 'Robin', 'Sneha', 'Ravi']

References

[1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?

Mutable and Immutable Objects

Lecture 7

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- In Python, ‘mutable’ is the ability of objects to change their values.
- In Python, if the value of an object cannot be changed over time, then it is known as immutable.
- List, Set and Dictionary are mutable.
- Strings and Tuples are immutable.
- Elementary forms of data such as numeric and Boolean are called scalar data types.
- Several applications require more complex forms of data, for example, the name of a person, coordinates of a point, a set of objects, or a list of personal records of individuals.

Lists

- A list is an ordered sequence of values. It is a non-scalar type.
- Values stored in a list can be of any type such as string, integer, float, or list, for example, a list may be used to store the names of subjects:
  ⟩⟩⟩ subjects=['Hindi', 'English', 'Maths', 'History']
- The elements of a list are enclosed in square brackets, separated by commas.
- Elements of a list are arranged in a sequence beginning index 0, just like characters in a string.

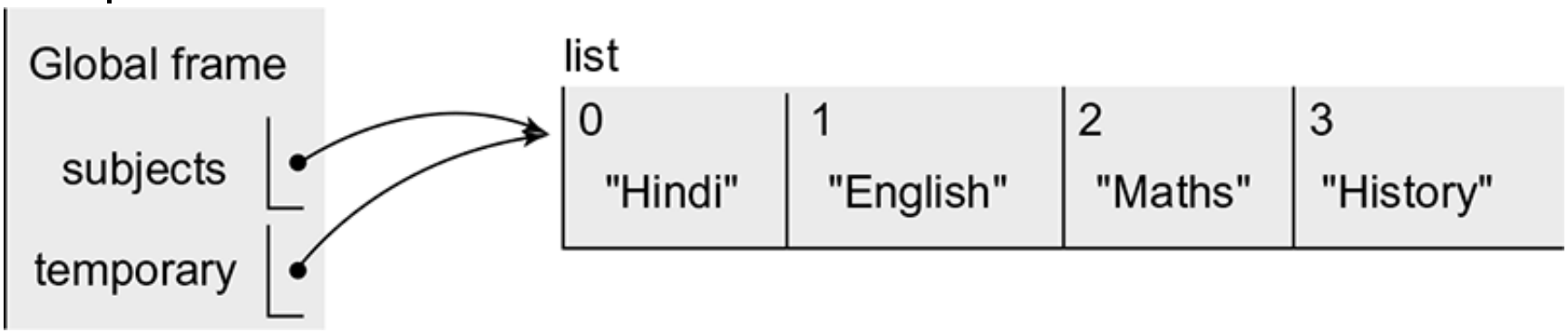
Lists (Cont.)

- We show the representation of the list subjects as in PythonTutor.

- To understand the lists better, let us invoke the function id that returns object identifier for the list object subjects:
  ⟩⟩⟩ id(subjects)
  57135752
- Different variables may refer to the same list object
  ⟩⟩⟩ temporary = subjects
  ⟩⟩⟩ id(temporary)
  57135752

Lists (Cont.)

- Each of the names subjects and temporary is associated with the same list object having object id 57135752.
- PythonTutor representation of the lists subjects and temporary, at this point, is shown below:



- This method of accessing an object by different names is known as aliasing.

Lists (Cont.)

- As each of the two names temporary and subjects refers to the same list, on modifying the component temporary[0] of the list temporary, the change is reflected in subjects[0] as shown below:
  ⟩⟩⟩ temporary[0] = 'Sanskrit'
  ⟩⟩⟩print(temporary)
  ['Sanskrit', 'English', 'Math', 'History']
  ⟩⟩⟩ print(subjects)
  ['Sanskrit', 'English', 'Math', 'History']
  ⟩⟩⟩ print(id(subjects), id(temporary))
  57135752 57135752

Lists (Cont.)

- **Two-dimensional list: list of lists**
- we create a list of subjects and their corresponding subject codes. For this purpose, we represent each pair of subject and subject code as a list, and form a list of such lists:
  ⟩⟩⟩ subjectCodes = [['Sanskrit', 43], ['English', 85] , ['Maths', 65], ['History', 36]]
- A list of lists such as subjectCodes, each of whose elements itself is a list, is called a two-dimensional list.
- Thus, subjectCodes[1] being a list, its components may be accessed as subjectCodes[1][0] and subjectCodes[1][1]:
  ⟩⟩⟩ subjectCodes[1]
  ['English', 85]
  ⟩⟩⟩ print(subjectCodes[1][0], subjectCodes[1][1])
  English 85

Lists (Cont.)

- Often times, we need to take a list as an input from the user. For this purpose, we use the function **input** for taking the input from the user, and subsequently apply the function **eval** for transforming the raw string to a list:
  ⟩⟩⟩ details = eval(input('Enter details of Megha: '))
  Enter details of Megha: ['Megha Verma', 'C-55, Raj Nagar,Pitam Pura, Delhi - 110034', 9876543210]
  ⟩⟩⟩ details
  ['Megha Verma', 'C-55, Raj Nagar,Pitam Pura, Delhi - 110034', 9876543210]

Lists (Cont.)

- we describe some functions and operations on lists with the help of the following lists:
  ⟩⟩⟩ list1 = ['Red', 'Green']
  ⟩⟩⟩ list2 = [10, 20, 30]

Lists (Cont.)

Operation	Example
Multiplication Operator $*$	$\rangle\rangle\rangle$ list2 $*$ 2 [10, 20, 30, 10, 20, 30]
Concatenation Operator $+$	$\rangle\rangle\rangle$ list1=list1 $+$ ['Blue'] $\rangle\rangle\rangle$ list1 ['Red', 'Green', 'Blue']
Length Operator **len**	$\rangle\rangle\rangle$ len(list1) 3
Indexing	$\rangle\rangle\rangle$ list2[-1] 30
Slicing Syntax: start:end:inc	$\rangle\rangle\rangle$ list2[0:2] [10, 20] $\rangle\rangle\rangle$ list2[0:3:2] [10, 30]
Function **min**	$\rangle\rangle\rangle$ min(list2) 10

Lists (Cont.)

Operation	**Example**
Function **max**	⟩⟩⟩ max(list1) 'Red'
Function **sum**(Not defined on strings)	⟩⟩⟩ sum(list2) 60
Membership operator **in**	⟩⟩⟩ 40 in list2 False

Table 1: Summary of operations that can be applied on lists

Lists (Cont.)

- The membership operator **in** may be used in a for loop for sequentially iterating over each element in the list, for example: iterating over the elements of a list

```
⟩⟩⟩ students = ['Ram', 'Shyam', 'Gita', 'Sita']
⟩⟩⟩ for name in students:
        print(name)
Ram
Shyam
Gita
Sita
```

Lists (Cont.)

- We list some important built-in functions that can be applied to lists.
- Many of these functions such as **append, reverse, sort, extend, pop, remove,** and **insert** modify the list.
- Functions such as **count** and **index** do not modify the list.
- **dir(list)** outputs the list including all the functions that can be applied to objects of the type **list**.

Lists (Cont.)

Function	Explanation
L.append(e)	Inserts the element e at the end of list L.
L.extend(L2)	Inserts the item in the sequence L2 at the end of elements of the list L.
L.remove(e)	Removes the first occurrence of the element e from the list L
L.pop(i)	Returns the elements from the list L at index i, while removing it from the list.
L.count(e)	Returns count of occurrences of object e in the list L.
L.index(e)	Returns index of an object e, if present in list.
L.insert(i,e)	Inserts element e at index i in list.
L.sort()	Sorts the elements of list.
L.reverse()	Reverses the order of elements in the list.

Table 2: List functions

Lists (List Comprehension)

- List comprehension provides a shorthand notation for creating lists.
- Suppose we want to create a list that contains cubes of numbers ranging from 1 to 10.
- To do this, we may create an empty list, and use a for-loop to append the elements to this list:

```
⟩⟩⟩ cubes =[]
⟩⟩⟩ end = 10
⟩⟩⟩for i in range(1, end + 1):
        cubes.append(i ** 3)
⟩⟩⟩ cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Lists (List Comprehension)

- Alternatively, a simple one line statement can be used for achieving the same task.
  $\rangle\rangle\rangle$ cubes = [x**3 for x in range(1, end + 1)]

Lists (List as Arguments)

```
1  def listUpdate(a, i, value):
2      '''
3      Objective: To change a value at a particular index in list
4      Input Parameters:
5      a - list
6      i - index of the object in the list to be updated
7      value - modified value at index i
8      Return value: None
9      '''
10     a[i] = value
11 def main():
12     '''
13     Objective: To change a value at a particular index in list
14     Input Parameters: None
15     Return value: None
16     '''
17     lst = [10, 20, 30, [40, 50]]
18     listUpdate(lst, 1, 15)
19     print(lst)
20 if __name__=='__main__':
21     main()
```

Lists (List as Arguments)

- The function listUpdate updates the list **a** by replacing the object a[i] by value.
- The main function invokes the function listUpdate with the arguments lst, 1, and 15 corresponding to the formal parameters a, i, and value.
- As arguments are passed by reference, during execution of the function listUpdate, an access to the formal parameter **a** means access to the list **lst** created in the main function.
- Consequently, when we update the list **a** in the function listUpdate, it results in the corresponding update of the list **lst**.
- Thus, the value at index 1 of the list **lst** gets updated to the value 15.

Lists (Copying List Objects)

- assigning a list to another name does not create another copy of the list, instead it creates another reference
  ⟩⟩⟩ list1 = [10, 20, [30, 40]]
  ⟩⟩⟩ list2 = list1
- As the names list1 and list2 refer to the same list object, any changes made in the list will relate to both the names list1 and list2, for example:
  ⟩⟩⟩ list1[1] = 22
  ⟩⟩⟩ list1
  [10, 22, [30, 40]]
  ⟩⟩⟩ list2
  [10, 22, [30, 40]]

Lists (Copying List Objects)

- To create another instance of the list object having different storage, we need to import the copy module and invoke the function **copy.copy()**:
  ⟩⟩⟩ import copy
  ⟩⟩⟩ list1 = [10, 20, [30, 40]]
  ⟩⟩⟩ list3 = copy.copy(list1)
- the copy function creates a new copy list3 of the list1. Consequently, on modifying list1[1], list3[1] remains unaltered:
  ⟩⟩⟩ list1[1] = 25
  ⟩⟩⟩ list1
  [10, 25, [30, 40]]
  ⟩⟩⟩ list3
  [10, 20, [30, 40]]

Lists (Copying List Objects)

- However, the copy function creates a shallow copy i.e. it does not create copies of the nested objects. Thus, the two lists share the same nested objects.
- list1[2] and list3[2] refer to the same nested list [30, 40].
- let us modify list1[2][0] in sub-list list1[2] of list list1 to value 35.

```
⟩⟩⟩ list1[2][0] = 35
⟩⟩⟩ list1
[10, 25, [35, 40]]
⟩⟩⟩ list3
[10, 20, [35, 40]]
```

Lists (Copying List Objects)

- To create a copy of a list object so that the nested objects (at all levels) get copied to new objects, we use the function deepcopy of the copy module:
- **deepcopy()**: To create a copy of a list including copies of the nested objects at all level

```
⟩⟩⟩ import copy
⟩⟩⟩ list1 = [10, 20, [30, 40]]
⟩⟩⟩ list4 = copy.deepcopy(list1)
⟩⟩⟩ list1[2][0] = 35
⟩⟩⟩ list1
[10, 20, [35, 40]]
⟩⟩⟩ list4
[10, 20, [30, 40]]
```

Lambda Expression

- A lambda expression consists of the lambda keyword followed by a comma separated list of arguments and the expression to be evaluated using the list of arguments in the following format:
- **lambda arguments: expression**
- Example: lambda expression to compute the cube of a number

```
⟩⟩⟩ cube = lambda x: x ** 3
⟩⟩⟩ cube(3)
27
```

- Example: lambda expression to compute sum of cubes of two numbers

```
⟩⟩⟩ sum2Cubes = lambda x, y: x**3 + y**3
⟩⟩⟩ sum2Cubes(2, 3)
35
```

map, reduce, and filter Operations on a Sequence

- Python provides several built-in functions based on expressions, which work faster than loop-based user defined code.
- The function map is used for transforming every value in a given sequence by applying a function to it.
- It takes two input arguments: the iterable object (i.e. object which can be iterated upon) to be processed and the function to be applied, and returns the map object obtained by applying the function to the list as follows:
- **result = map(function, iterable object)**
- The function to be applied may have been defined already, or it may be defined using a lambda expression which returns a **function** object.

map, reduce, and filter Operations on a Sequence

- Mapping every value in the sequence to its cube
  ⟩⟩⟩ lst = [1, 2, 3, 4, 5]
  ⟩⟩⟩ list(map(lambda x: x ∗∗ 3, lst))
  [1, 8, 27, 64, 125]
  ⟩⟩⟩ list(map(cube, lst))
  [1, 8, 27, 64, 125]
- We may also use any system defined or user-defined function as an argument of the map function, for example:
  ⟩⟩⟩ list(map(abs, [-1, 2, -3, 4, 5]))
  [1, 2, 3, 4, 5]

map, reduce, and filter Operations on a Sequence

- Suppose we wish to compute the sum of cubes of all elements in a list.
- Adding elements of the list is a repetitive procedure of adding two elements at a time.
- The function reduce, available in functools module, may be used for this purpose. It takes two arguments: a function, and a iterable object, and applies the function on the iterable object to produce a single value:

```
⟩⟩⟩ lstCubes = list(map(lambda x: x ** 3,lst))
⟩⟩⟩ import functools
⟩⟩⟩ sumCubes = functools.reduce(lambda x, y: x + y, lstCubes)
⟩⟩⟩ sumCubes
225
```

map, reduce, and filter Operations on a Sequence

- To compute the sum of cubes of only those elements in the list lstCubes that are even.
- Thus, we need to **filter** the even elements from the list lstCubes
- Python provides the function **filter** that takes a function and a iterable object as the input parameters and returns only those elements from the iterable object for which function returns True.
- Since the function **filter** returns a **filter** object, we have used the function **list** to convert it to list.

map, reduce, and filter Operations on a Sequence

```
⟩⟩⟩ evenCubes = list(filter(lambda x: x%2== 0, lstCubes))
⟩⟩⟩ evenCubes
[8, 64]
⟩⟩⟩ sumEvenCubes = functools.reduce(lambda x, y: x + y,
evenCubes)
⟩⟩⟩ sumEvenCubes
72
```

- Alternatively, the sum of odd cubes may be computed as follows:

```
⟩⟩⟩ sumEvenCubes = functools.reduce(lambda x, y: x + y,
filter(lambda x: x%2 == 0, lstCubes))
⟩⟩⟩ sumEvenCubes
72
```

- The functions map, reduce, and filter are often used to exploit the parallelism of the system to distribute computation to several processors.

Sets

- A comma separated unordered sequence of values enclosed within curly braces is called **Set**.
- Function **set()** is used to convert a sequence to a set.

```
>>> vowels = set('aeiou')
>>> vowels
{'e', 'a', 'o', 'u', 'i'}
>>> vehicles = set(['Bike', 'Car', 'Bicycle', 'Scooter'])
>>> vehicles
{'Scooter', 'Car', 'Bike', 'Bicycle'}
>>> digits = set((0, 1, 2, 3, 4, 5, 6, 7,8, 9))
>>> digits
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Sets

- Elements of a set must be immutable.
- Set type does not support indexing, slicing, + operator, and * operator.
- We can iterate over elements of the set using **in** operator

```
⟩⟩⟩ for v in vowels:
        print(v, end = ' ')
e a o u i
```

- The functions **min, max, sum**, and **len** work for sets in the same manner as defined for lists.

```
⟩⟩⟩ len(vehicles)
4
```

- The membership operator **in** checks whether an object is in the set.

```
⟩⟩⟩ 'Bike' in vehicles
True
```

Sets (Functions)

- List of some important built-in functions that can be applied to sets.

Function	**Description**
S.add(e)	Adds the elements to the set S, if not present already.
S1.update(L1)	Add the items in object L1 to the set S1, if not already present.
S.remove(e)	Removes the element e from set S.
S.pop()	Removes an element from the set S.
S.clear()	Removes all elements from the set S.
S.copy()	Creates a copy of the set S.
S1.union(S2)	Returns union of the Sets S1 and S2.
S1.intersection(S2)	Returns a set containing common elements of sets S1 and S2.

Sets (Functions)

- List of some important built-in functions that can be applied to sets.

Function	**Description**
S1.difference(S2)	Returns a set containing elements in set S1 but not in set S2.
S1.symmetric_difference(S2)	Returns a set containing elements that are in one of the two sets S1 and S2, but not in both.

Table 3: Set Functions

- The operators $<=$, $==$, and $>=$ may be used to check whether a given set is a subset, equal, or superset of another set.

Sets (Set Comprehension)

- Set comprehension is applied to find common factors
- A common factor cannot exceed smaller of the two numbers, say n1 and n2.
- To build a set of common factors, we make use of comprehensions. For each number i in range(1, min(n1,n2) + 1), we include it in the set if it is a factor of each of n1 and n2.
- Thus, our code comprises just one line:

```
1 commonFactors = {i for i in range(1,min(n1+1, n2+1)) if n1%
      i == 0 and n2%i ==0}
```

Tuples

- A **tuple** is a non-scalar type defined in Python. Just like a list, a tuple is an ordered sequence of objects.
- However, unlike **lists**, **tuples** are immutable, i.e. elements of a **tuple** cannot be overwritten.
- A tuple may be specified by enclosing in the parentheses, the elements of the tuple (possibly of heterogeneous types), separated by commas, for example, the tuple t1 comprises five objects:
  ⟩⟩⟩ t1 = (4, 6, [2, 8], 'abc', {3,4})
  ⟩⟩⟩ type(t1)
  ⟨*class' tuple'*⟩

Tuples

- If a tuple comprises a single element, the element should be followed by a comma to distinguish a tuple from a parenthesized expression, for example:
  ⟩⟩⟩ (2,)
  (2,)
- A tuple having a single element is also known as singleton tuple.
- Another notation for tuples is just to list the elements of a tuple, separated by commas:
  ⟩⟩⟩ 2, 4, 6
  (2, 4, 6)
- Elements of a tuple may be mutable
  ⟩⟩⟩ t1 = (1, 2, [3, 4])
  ⟩⟩⟩ t1[2][1] = 5
  ⟩⟩⟩ t1
  (1, 2, [3, 5])

Tuple Operations

- we summarize the operations on tuples and use the following tuples t1 and t2 for the purpose of illustration:
  ⟩⟩⟩ t1 = ('Monday', 'Tuesday')
  ⟩⟩⟩ t2 = (10, 20, 30)

Operation	**Example**
Multiplication Operator $*$	⟩⟩⟩ t1 $*$ 2 ('Monday', 'Tuesday', 'Monday', 'Tuesday')
Concatenation Operator $+$	⟩⟩⟩ t3=t1 $+$ ('Wednesday',) ⟩⟩⟩ t3 ('Monday', 'Tuesday', 'Wednesday')
Length Operator **len**	⟩⟩⟩ len(t1) 2
Indexing	⟩⟩⟩ t2[-2] 20

Tuple Operations

Operation	**Example**
Slicing Syntax: start:end:inc	⟩⟩⟩ t1[1:2] ('Tuesday',)
Function **min**	⟩⟩⟩ min(t2) 10
Function **max**	⟩⟩⟩ max(t2) 30
Function **sum** (not defined on strings)	⟩⟩⟩ sum(t2) 60
Membership operator **in**	⟩⟩⟩ 'Friday' in t1 False

Table 4: Summary of operations that can be applied on tuples

Functions Tuple and Zip

- The function tuple can be used to convert a sequence to a tuple, for example:
  ⟩⟩⟩ vowels = 'aeiou'
  ⟩⟩⟩ tuple(vowels)
  ('a', 'e', 'i', 'o', 'u')
- The function zip is used to produces a zip object (iterable object), whose ith element is a tuple containing ith element from each iterable object passed as argument to the zip function.
- We have applied list function to convert the zip object to a list of tuples. For example,
  ⟩⟩⟩ colors = ('red', 'yellow', 'orange')
  ⟩⟩⟩ fruits = ['cherry', 'banana', 'orange']
  ⟩⟩⟩ fruitColor = list(zip(colors, fruits))
  ⟩⟩⟩ fruitColor
  [('red', 'cherry'),('yellow', 'banana'),('orange', 'orange')]

Functions count and index

- The function **count** is used to find the number of occurrences of a value in a tuple, for example:
  ⟩⟩⟩ age = (20, 18, 17, 19, 18, 18)
  ⟩⟩⟩ age.count(18)
- The function **index** is used to find the index of the first occurrence of a particular element in a tuple, for example:
  ⟩⟩⟩ age.index(18)
  1

Function	Explanation
T.count(e)	Returns count of occurrences of e in Tuple T
T.index(e)	Returns index of first occurrences of e in Tuple T

Table 5: Tuple Functions

Dictionary

- Unlike lists, tuples, and strings, a **dictionary** is an unordered sequence of key-value pairs.
- Indices in a **dictionary** can be of any immutable type and are called keys.
- Beginning with an empty dictionary, we create a dictionary of month_number-month_name pairs as follows:

```
⟩⟩⟩ month = {}
⟩⟩⟩ month[1] = 'Jan'
⟩⟩⟩ month[2] = 'Feb'
⟩⟩⟩ month[3] = 'Mar'
⟩⟩⟩ month[4] = 'Apr'
⟩⟩⟩ month
{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr'}
⟩⟩⟩ type(month)
⟨class'dict'⟩
```

Dictionary

```
⟩⟩⟩ price = {'tomato':40, 'cucumber':30, 'potato':20,
'cauliflower':70, 'cabbage':50, 'lettuce':40, 'raddish':30, 'carrot':20,
'peas':80}
⟩⟩⟩ price['potato']
20
⟩⟩⟩ price['carrot']
20
⟩⟩⟩ price.keys()
dict_keys(['tomato', 'cucumber', 'potato', 'cauliflower', 'cabbage',
'lettuce', 'raddish', 'carrot', 'peas'])
⟩⟩⟩ price.values()
dict_values([40, 30, 20, 70, 50, 40, 30, 20, 80])
⟩⟩⟩ price.items()
dict_items([('tomato', 40), ('cucumber', 30), ('potato', 20),
('cauliflower', 70), ('cabbage', 50), ('lettuce', 40), ('raddish', 30),
('carrot', 20), ('peas',80)])
```

Dictionary

- The search in a dictionary is based on the key. Therefore, in a dictionary, the keys are required to be unique.
- As keys in a dictionary are immutable, lists cannot be used as keys.
- Keys in a dictionary may be of heterogeneous types, for example:
  ⟩⟩⟩ counting = {1:'one', 'one':1, 2:'two', 'two':2}

Dictionary Operations

- some operations that can be applied to a dictionary and illustrate these operations using a dictionary of digit-name pairs:
  digits = {0:'Zero', 1:'One', 2:'Two', 3:'Three', 4:'Four', 5:'Five', 6:'Six', 7:'Seven', 8:'Eight', 9:'Nine'}

Operation	**Examples**
Length operator **len** (number of key-value pairs in dictionary)	⟩⟩⟩ len(digits) 10
Indexing	⟩⟩⟩ digits[1] 'One'
Function **min**	⟩⟩⟩ min(digits) 0
Function **max**	⟩⟩⟩ max(digits) 9
Function **sum** (assuming keys are compatible for addition)	⟩⟩⟩ sum(digits) 45

Dictionary Operations

Operation	**Examples**
Membership operator **in**	$\rangle\rangle\rangle$ 5 in digits True $\rangle\rangle\rangle$ 'Five' in digits False

Table 6: Summary of operations that can be applied on dictionaries

- Consider a dictionary named as winter:
  $\rangle\rangle\rangle$ winter = {11:'November ', 12: 'December', 1:'January', 2:'February'}
- Membership operation **in**, and functions **min**, **max** and **sum** apply only to the keys in a dictionary.

Dictionary Operations

- Thus, applying these operations on the dictionary winter is equivalent to applying them on winter.keys() as shown below:
  ⟩⟩⟩ 2 in winter, min(winter), max(winter), sum(winter)
  (True, 1, 12, 26)
  ⟩⟩⟩ 2 in winter.keys(), min(winter.keys()), max(winter.keys()), sum(winter.keys())
  (True, 1, 12, 26)
- We may remove a key-value pair from a dictionary using del operator, for example:
  ⟩⟩⟩ del winter[11]
  ⟩⟩⟩ winter
  {12: 'December', 1: 'January', 2: 'February'}
- To delete a dictionary, we use del operator:
  ⟩⟩⟩ del winter

Dictionary Functions

Function	Explanation
D.items()	Returns an object comprising of tuples of **key-value** pairs present in dictionary D
D.keys()	Returns an object comprising of all keys of dictionary D
D.values()	Returns an object comprising of all values of dictionary D
D.clear()	Removes all **key-value** pairs from dictionary D
D.get(key, default)	For the specified **key**, the function returns the associated value. Returns the **default** value in the case **key** is not present in the dictionary D
D.copy()	Creates a shallow copy of dictionary D
D1.update(D2)	Adds the **key-value** pairs of dictionary D2 to dictionary D1

Table 7: Dictionary Functions

Inverted Dictionary

- Suppose we are maintaining a dictionary of words and their meanings of the form **word:meaning**. For simplicity, we assume that each word has a single meaning.
- There might be a few words that may have shared meaning. Given this dictionary, we wish to find synonyms of a word, i.e. given a word, we wish to find the list of words that have the same meaning.
- For this purpose, we would like to build an inverted dictionary **invDict** of **meaning:list-of-words**.
  ⟩⟩⟩Enter word meaning dictionary: {'dubious':'doubtful', 'hilarious':'amusing', 'suspicious':'doubtful', 'comical':'amusing', 'hello':'hi'}
  Inverted Dictionary:
  {'doubtful': ['dubious', 'suspicious'], 'amusing': ['hilarious', 'comical']}

Inverted Dictionary Code

```
1  def buildInvDict(dict1):
2      '''
3      Objective: To construct inverted dictionary
4      Input parameter: dict1: dictionary
5      Return value: invDict: dictionary
6      '''
7      invDict = {}
8      for key,values in dict1.items():
9          if value in invDict:
10             invDict[value].append(key)
11         else:
12             invDict[value] = [key]
13     invDict={x:invDict[x] for x in invDict if len(invDict[x]>1)
       }
14     return invDict
```

Inverted Dictionary Code

```
1  def main():
2      '''
3      Objective: To find inverted dictionary
4      Input parameters: None
5      Return value: None
6      '''
7      wordMeaning = eval(input('Enter word meaning dictionary:'))
8      meaningWord = buildInvDict(wordMeaning)
9      print('Inverted dictionary:\n', meaningWord)
10
11 # Statements to initiate the call to main function.
12
13 if __name__=='__main__':
14     main()
```

References

Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?