

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 1: 04/10/2021

Dr. Debranjan Sarkar

Installing Python

Installing Python

- Python is available on various OS platforms viz. Window, Linux, MacOS etc.
- There are different versions of Python, most popular ones are
 - Python 2.7.x
 - Python 3.x.y
- It is seen from the site www.python.org that Python 3.9.7 is the latest version for Windows
- Note that Python 3.9+ cannot be used on Windows 7 or earlier
- Python 2.7 is an older version, but still alive as many libraries for scientific and statistical computation are still in Python 2.7
- We shall work with Python 3.x.y

Installing Python

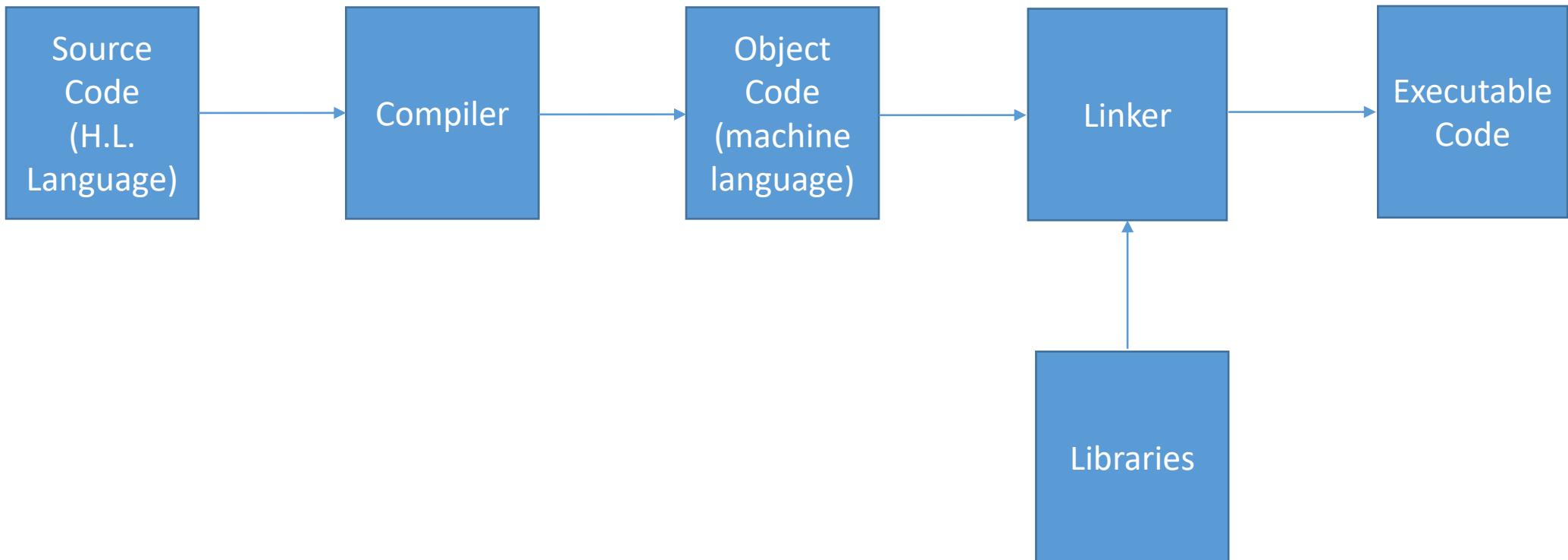
- Use the site <https://www.python.org/downloads> to download and install the latest version of Python on your PC on different OS Platforms:
 - Windows
 - Linux / UNIX
 - MacOS and others....
- Now, you should also install a Python IDE (Integrated Development Environment) and Code Editor
- Various Python IDEs and Code Editors are available
- One such Code Editor (PyCharm) is available for download and installation from the site <https://www.jetbrains.com/PyCharm/>
- Install the IDE and Code Editor on your PC
- Now you are able to write/edit a Python Program and execute it once you know how to code in Python language

The Python Language

What is Python?

- Python is a high level programming language like C, C++, etc. to facilitate human being to write codes in an English-like language
- However, computers understand only low level or machine language
- Compiler translates high level language to machine language and after linking with different libraries, executable code is generated
- Interpreter interprets /understands each and every line (one by one) of a program written in high level language and executes it, if required
- Python is basically an interpreted language
 - Python commands are input to the interpreter for execution
 - With interpreter, it is easy to interactively explore the language features
 - Python interpreter can load complex programs from files
 - >>> from filename import *

Source Code to Executable Code



First few Python statements

- How to display a text?
- print ("Hello students of HITK")
- 'print' statement automatically generates a new line (\n) after displaying the text
- How to give comment in a program?
- Hash symbol (#) is used for a single line comment and Python ignores the comment line
- 3 numbers of single quote (') or double quote ("""") pairs are used for multi-line comments as shown below:
- """
- Comment line number 1
- Comment line number 2
- Comment line number 3
- """

First few Python statements

- `s = input('Your name: ')` # It prompts for typing your name
- `print('Hello ' + s + ', How are you?')` # Concatenation
- `print("I like Tagore's song.")` # Single and Double Quote
- `x = 9` # value 9 is assigned to variable x
- `print (x)` # prints the value of x
- `y = 10` # value 10 is assigned to variable y
- `z = x + y` # The sum of x and y is assigned to z
- `print ("Sum = ", z)`
- `A = 10`
- `a = 5`
- `print('A = ', A, 'a = ', a)` # Small letters and capital letters are different

Datatypes

- Information about a student:
 - (Name) (Surname) (Roll No.) (Age) (Height) (whether SC/ST)
- Concept of names, values and types
- Values have types which determine which operations are allowed
- Different datatypes are possible in Python
 - String 'Heritage Institute of Technology'
 - If there is a single quote in the string “ may be used (\' may also be used)
 - If both single quote and double quote are there, triple single or double quote may be used
 - All the following examples of string datatype are correct
 - “I like Tagore’s song”
 - ‘I like Tagore\’s song’
 - """Ram’s son said “Hello” to me””
 - """”Ram’s son said “Hello” to me”””
 - Integer 25, -7
 - No decimal point
 - Binary representation with sign bit
 - 2’s complement for negative numbers

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 2: 05/10/2021

Dr. Debranjan Sarkar

Datatypes

- Float 3.14, 99.6
 - Decimal point in the number
 - IEEE 754 single or double precision representation (mantissa and exponent part)
 - float is a generalized form of int 3 -> 3.0
- Bool True, False
 - First character must be in capital (true and false are incorrect)
- Function ‘type(expression)’ may be used to know the type of a data
 - `print(type(5.7))` will produce output as <class ‘float’>
 - `print(type(5 == 7))` will produce output as <class ‘bool’>

Names (Variables)

- A name can be associated with a value by way of assignment
- A named value is termed as variable as its value may change over time
- Name must begin with a letter or underscore and can include any number of letters, digits or underscores only and no other characters (e.g. space, \$. % etc.)
- Name cannot begin with a digit
- Small letter and capital letter are different in Python
- `x = 9` => Here x is the name of a variable and 9 is a constant
- `z = x + y` => Here, x, y and z are names of variables
- `x = x + 1` => Here = is an assignment operator

Names (Variables)

- Variables need not be explicitly defined before using it (unlike C language)
- Names inherit type from currently assigned value
 - If $x = 5 \Rightarrow x$ is int $x = \text{True} \Rightarrow x$ is bool
- Names do not have a fixed type
- Values of different types can be assigned to a name
- Without assigning a value to a variable, it cannot be used
- The programmer should be careful in choosing name of a variable
 - Example: a good programmer would prefer to use ‘student_name’ as the name of the variable Student’s name rather than ‘name’ or ‘nofs’ etc. so that it carries meaning to the others who look into the program

Operators

- **Binary Operators**
 - + - * ** / // % << >> & | ^ < > <= >= == != <>
 - / always produces a floating point number
 - Example: $9 / 3 = 3.0$ $9/4.5 = 2.0$ $9.5/1.9 = 5.0$
 - $16 // 3 = 5$ (quotient) and $16 \% 3 = 1$ (remainder or modulus)
 - $14 // 7 = 2$
 - **Unary operators**
 - + - ~ \sim True = False $a = -b$
 - **Operator precedence**
 - () ** * / // % + - from left to right
 - $2+3*4 \rightarrow 14$ $(2+3)*4 \rightarrow 20$
 - **Logical operators**
 - and, or, not $(5 < 10) \text{ and } (3 > -1) \rightarrow \text{True}$

Expression and Statement

- Expression is any section of the code that evaluates to a value
 - Arithmetic: $9 + 5 * 2$
 - Boolean or conditional: $5 \leq 10$
 - Expressions may contain a mixed number and floating-point components. Example:
 - `x = 4` # int
 - `y = 10.2` # float
 - `total = x + y` # mixed
 - Here, variable total is of type float
- A Statement is a complete line of code that performs some action
 - Assignment statement (assign a value to a name): $x = 9 + 5 * 2$
 - Simultaneous assignment $(x,y) = (2,3)$
 - Print statement: `print ("Hello, How are you?")`

Types of values in Python (Numeric, Logical)

- Values have types that determine which operations are allowed
- Numeric type
 - Numeric type can be integer (int) or floating point (float) numbers
 - Allowed Arithmetic operations: + - * ** / etc.
- Logical values (bool) {True, False}
 - Logical operations: and or not etc.
 - Comparison: == <= >= <> != etc.
 - Combination: (x == y) and (s >= t)

Types of values in Python (Strings)

- String (str) is a sequence of characters Example: `s = 'Technology'`
- A single character is a string of length 1 (No separate 'char' type as in C)
- Values can be extracted by position Example: `s[0] = 'T'` `s[-1] = 'y'`
- Slice or substring can be extracted as follows:
 - `s[2:4] = 'ch'` `s[0:4] = 'Tech'`
- Strings can be concatenated (+ operator)
 - Example: `t = "Her" + "itage"`
- Whether a string is a part of another string can be found by 'in' operator
 - 'n' in s => True 'u' in s => False
- Length of a string is given by the function 'len'
 - Example: `len(s) = 10`
- There are many other functions

Types of values in Python (Lists)

- List is a sequence of values:
 - numbers = [4,8,12,16] List of integer numbers
 - names = ['Tom', 'Dick', 'Harry'] List of strings
 - mixed-type-list = ['Tom', 12, True] List of values of mixed (non-uniform) types
- Like string, Values can be extracted by position
 - numbers[0] = 4
 - names[1:] = ['Dick', 'Harry']
- Like string, Lists can be concatenated (+ operator)
 - list1 = [1,3,5,7,9] list2 = [3,6,9] list3 = list1 + list2 => list3 = [1,3,5,7,9,3,6,9]
 - After concatenation, always a new list is produced
 - list1 = [2,4,6]
 - list2 = list1 list1 and list2 point to the same object list1 is list2 => True
 - list1 = [1,3,5] After this statement is executed, list2 also becomes [1,3,5]
 - list1 = list1 + [8] list1 and list2 no longer point to the same object list1 is list2 => False
- Like string, Length of a list is given by the function 'len'
 - len(numbers) = 4

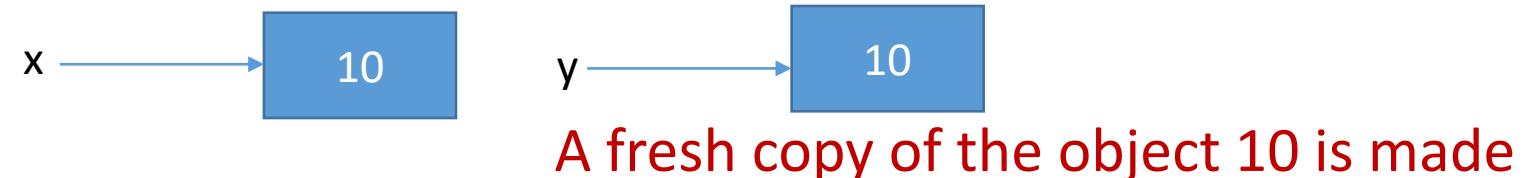
What is an immutable data type?

- Immutable data types are the objects that cannot be modified and altered (i.e. adding new elements, removing an element or replacing an element), after it is created (Examples of immutable data type: int, float, bool, str etc.)

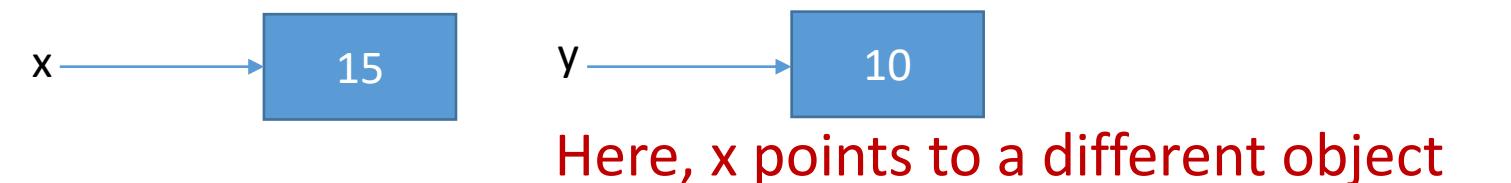
- $x = 10$



- $y = x$



- $x = 15$

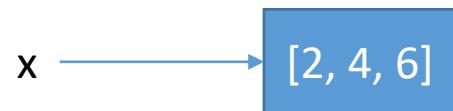


- So, the value of x is 15, but the value of y remains 10

What is a mutable data type?

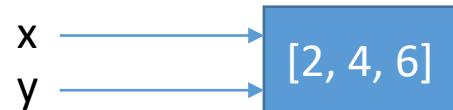
- Mutable data types are the objects that can be changed after it is created
- Examples of mutable data type: list etc.

- $x = [2, 4, 6]$



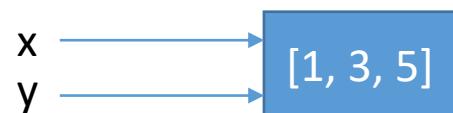
x points to the object $[2, 4, 6]$

- $y = x$



y points to the same object $[2, 4, 6]$

- $x = [1, 3, 5]$



y also becomes $[1, 3, 5]$

- Here, the object itself has changed
- As both x and y point to the same object, which itself has been changed, the values of both x and y become $[1, 3, 5]$

Difference between mutable and immutable values

- Consider the following assignment statements:

- `x = 10` # Type: int
- `y = x`
- `x = 20`

- What will be the value of y after execution of the above statements?
- For immutable values (Example: Values of type int, float, bool, str):
 - Assignment does make a fresh copy
 - The 2nd statement copies the current value of x for y, then the value of y remains 10 even after execution of the 3rd statement
 - Updating one value does not affect the copy

Difference between mutable and immutable values

- Consider the following assignment statements:
 - `x = [10, 20, 30, 40]` # Type: list
 - `y = x`
 - `x[2] = 35`
- What will be the value of y after execution of the above statements?
- For mutable values (Example: Values of type list):
 - Assignment does not make a fresh copy
 - The 2nd statement does not make a fresh copy of the current value of x.
 - Rather, both the names x and y point to the same value
 - After execution of the 3rd statement, the value of x changes to [10, 20, 35, 40]
 - So y will also become [10, 20, 35, 40], as x and y are the two names of the same list
 - To make a copy of a list we use full slice Example: `y = x[:]`

Strings are immutable

- Strings are immutable that is strings cannot be updated “in place”
- Python does not allow to change part of a string
 - Example: `s = 'better'` we want to update the string as ‘bitter’
 - Python does not allow `s[1] = 'i'` => error!
- Use slices and concatenate these to get a new string
 - `s = s[:1] + 'i' + s[2:]`
 - This `s` is a new string and not the old string

Lists are mutable

- List is a mutable object
- Lists can be updated in-place (unlike strings)
 - Assigning a new value to a slice:
 - `lst1 = [3, 5, 7, 9]`
 - `lst2 = lst1`
 - `lst1[1:] = [6, 9, 12]`
- Slice replacement happens in-place
- Now `lst1` and `lst2` are both `[3,6,9,12]`
- Slices can be expanded or contracted in-place
 - `lst1[1:] = [6, 9, 12,15,18]` Now `lst1` and `lst2` are both `[3,6,9,12,15,18]`
 - `lst1[1:] = [6]` Now `lst1` and `lst2` are both `[3,6]`

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 3: 25/10/2021

Dr. Debranjan Sarkar

More on Lists

- Concatenation produces a new list
 - `list1 = [2,4,6,8]` `list2 = list1` `list1 = list1[0:2] + [10] + list1[3:]`
 - Now, `list2` will remain `[2,4,6,8]` whereas `list1` is `[2,4,10,8]`
- How to append a new item in the same list?
 - `list1 = [2,4,6,8]` `list1 = list1 + [10]` # To append `[10]` to the list
 - `[10]` is appended but to a new list so, `list2` is not the same as `list1`
 - `list1.append(10)` value 10 is appended in the same list (in-place)
 - ‘append’ takes a single value
 - For ‘append’, it is to be informed to Python that `list1` is of type list. `list1 = []`
 - To extend `list1` by a list of values we use `list1.extend(list2)`
 - `list1 = [1,3,5], list2 = [7,9]` `list1.extend(list2)` will produce `[1,3,5,7,9]`
 - ‘extend’ is in-place equivalent of concatenation
- Removal of item from the list
 - `list1.remove(v)` removes the first occurrence of `v` from the list
 - Generates error if there is no `v` in the list

More on Lists

- List membership: `(v in lst)` returns True if the value v is found in the list lst
- To remove v from lst, with no error, if v not found in lst
 - `if v in lst:`
`lst.remove(v)`
 - To remove all occurrences of v from lst
 - `while v in lst:`
`lst.remove(v)`
 - `lst.reverse()` # To reverse the list lst in-place
 - `lst.sort()` # To sort the list lst in ascending order
 - `lst.index(v)` # To find out the leftmost position of v in lst
Gives error if there is no v in lst
 - `lst.count(v)` # To count the number of occurrences of v in lst
 - And many more

Nested Lists

- An element of a list can be another list, which can have, in turn, another list and so on
- This type of list is called a nested list. Example:
 - `nested_list = [['Tom', 'Harry'], 24, [True], [5, [10,15]]]`
 - `nested_list[3] = [5, [10,15]]`
 - `nested_list[3][1] = [10,15]`
 - `nested_list[3][1][0] = 10`
 - `nested_list[0][0][1] = 'o'`
- Example of updating a nested list in-place (mutability)
 - `nested_list[1] = 25` is allowed
 - `nested_list` now becomes `[['Tom', 'Harry'], 25, [True], [5, [10,15]]]`
 - `nested_list[3][1][0] = 20`
 - `nested_list` now becomes `[['Tom', 'Harry'], 25, [True], [5, [20,15]]]`

Lists vs strings

- Both strings and lists can be indexed (or sliced)
- In case of string, single position as well as a slice return strings
 - college = 'Heritage' => college[0] is 'H' college[0:2] is 'He'
- For list, single position returns a value but a slice returns a list
 - numbers = [4,8,12,16] => numbers[2] = 12 numbers[2:3] = [12]
- Strings are immutable whereas lists are mutable
- Lists can be nested
- List can be converted into a string by '/'.join(['a', 'b', 'c']) => 'a/b/c'
- String can be converted into a list by list('xyz') => ['x', 'y', 'z']

Equality (`==`) and Identity (`is`)

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 4: 26/10/2021

Dr. Debranjan Sarkar

Type conversion

- `str(69) = "69"` # conversion from integer to string
- `int("854") = 854` # conversion from string to integer
- `int("3A") => Error` # string is not a decimal number
- `range(0,n)` generates the sequence 0, 1, 2, ..., n-1
- Sequence produced by the `range()` function is not of type list
- So we use **list function** to convert this sequence into type list
- `list(range(0,5)) = [0,1,2,3,4]`
- **list function** is also used to convert a string into list
- `list('abcdef')` will produce the list ['a', 'b', 'c', 'd', 'e', 'f']

Arrays

- A sequence of values can be stored as an array or as a list
- Features of an array:
 - Contiguous area of memory for storing the consecutive elements of an array
 - Elements are of uniform type (int/ float etc.) so that each element occupies same number of bytes (or words) in memory
 - Typically size of the sequence (number of elements) is fixed in advance
 - Array indexing is fast and accessing i^{th} sequence for all i , takes constant time
 - Offset is computed from the start of memory block
 - Word address of the 1st element of the array = s , each element requires n words (say)
 - Then the word address of the i^{th} element of the array is $(s + (i-1)*n)$
 - In the above example, $s = 57C$, $n = 2 \Rightarrow$ word address of the 4th element = $57C + (4-1)*2 = 582$
 - Insertion of an element between $\text{seq}[n]$ and $\text{seq}[n+1]$ is expensive
 - Deletion of an element in the sequence is also expensive

57C	A[0]
57E	A[1]
580	A[2]
582	A[3]
584	A[4]

Lists

57C	A[0]
57E	A[4]
580	A[2]
582	A[1]
584	A[3]

- **Features of a list:**

- Consecutive elements need not be stored in contiguous area of memory
- It is like data structure linked-list where in each element there is a pointer to the next element and for the last element, the pointer to the next element is Null
- Elements of a list may be non-uniform type (int/ string etc.) and have flexible size
- Size of the sequence (number of elements) is not to be fixed in advance, as the last element of a list contains a Null pointer to indicate that this is the last element
- To access the i^{th} sequence, i number of links have to be traversed and so the time is proportional to i
- Insertion and deletion of an element is easy and less expensive

Array vs List

- Exchange Operations (exchange the values of $\text{seq}[i]$ and $\text{seq}[j]$)
 - Takes constant time in array`
 - Takes linear time in list
- Delete $\text{seq}[i]$ or insert value v after $\text{seq}[i]$
 - Constant time in list as we are already at $\text{seq}[i]$ and we only have to shift some links
 - Linear time in array as the elements before or after the $\text{seq}[i]$ are to be shifted
- So the algorithms for a sequence of values represented as an array may not work well for data structure list

Control Flow : Conditionals

- Statements of a Python program are executed normally from top to bottom
- Control flow determines the order in which the statements are to be executed
- A common task is to choose between two or more alternative possibilities depending upon the outcome of a test
- Each alternative is called a conditional
- Conditional execution (if)

```
if condition:          # condition returns True or False, ':' indicates end of condition
    Statement1
    Statement2      # Uniform indentation demarcates the body of 'if'
    Statement3      # No curly brackets (braces) required as in C
```

- Statements 1 and 2 are executed when the condition is True
- Statement 3 is executed unconditionally (i.e. irrespective of whether condition is True or False)

Control Flow: Conditionals

- Conditional execution (if-else)

if condition:

 Statement1

.....

else:

 Statement2

.....

 Statement3

.....

- Statement 1 ... are executed when the condition is True
- Statement 2 ... are executed when the condition is False
- Shortcut for conditions
 - If a number is 0, it is as good as False
 - Empty sequence (Null string "" or empty list []) is treated as False
 - Everything else is True
 - Example: if m != 0: is equivalent to if m:

Control Flow: Conditionals

- Multiway branching (if-elif-else)

```
if condition1:
```

Statement1

.....

```
elif condition2:
```

Statement2

.....

```
elif condition3:
```

Statement3

.....

```
else:
```

Statement4

.....

Control Flow (Loops or Iteration)

- Repeat some statements for a fixed number of times

- `for i in [1,3,5,7,9]:`

```
    sq = i * i      # indentation for the body of the 'for' statement
```

```
    double = i * 2
```

```
    print('i = ', i, 'square = ', sq, 'double = ', double)
```

- `for i in [0,1,2,3,4,5,6,7,8,9]:`

- `for i in range(10):`

.....

.....

- `range(m,n)` produces the sequence $m, (m+1), (m+2), \dots, (n-1)$

- `range(n)` produces the sequence $0, 1, 2, \dots, (n-1)$

- `range(m,n,k)` produces the sequence $m, (m+k), (m+2k), \dots, (m + l.k)$ such that
$$m + lk < n \leq m + (l+1)k$$

- When k is negative, there will be a countdown

- `range(m,n,-1)` produces the sequence $m, (m-1), (m-2), \dots, (n + 1)$ $[m > n]$

More on range function

- Example: Find the sum of first 10 natural numbers
- n = 10
- sum = 0
- for i in range(1,n+1):
 - sum = sum + i
- print("sum = ", sum)
- =====
- # Compare range and list
- Is range(0,5) == [0,1,2,3,4] ?
- Answer: In Python 2 → YES In Python 3 → NO
- list(range(0,5)) == [0,1,2,3,4] in Python 3 # Type conversion

Control Flow (while)

- Number of iterations of a loop is not known in advance
- Repeat some statements until a condition becomes False
- `i = 0`
- `while i <= 9:`
 - `sq = i * i`
 - `double = i * 2`
 - `print('i = ', i, 'square = ', sq, 'double = ', double)`
 - `i = i + 1` # indentation for the body of the 'while' statement
- Condition should be made False inside the body of the statement, otherwise the loop will become an infinite loop

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 5: 27/10/2021

Dr. Debranjan Sarkar

'break' statement

- Sometimes we require to come out of the loop (both for loop and while loop) before the terminating condition is reached
- Write a function `find_position` to find the position of the first occurrence of a value `val` in a list `lst`. It returns -1 if there is no `val` in `lst`
- `def find_position(lst, val):`

```
    pos = -1
    for i in range(len(lst)):
        if lst[i] == val:
            pos = i
            break
    return(pos)
```
- How to detect whether the loop was terminated naturally or it came out of the loop through the `break` statement?

‘else’ after a for-loop or a while-loop

- ‘else’ statement after the for loop or while loop indicates that the loop (for or while) was terminated naturally
- def find_position(lst, val):

```
# pos = -1          this statement has been commented
for i in range(len(lst)):
    if lst[i] == val:
        pos = i
        break
else:
    # This statement is executed when the for loop was terminated naturally (i.e. val was
    # not found in lst
    pos = -1
return(pos)
```

What is a Function?

- Suppose a mathematical function $f(x)$ is given by
- | | | |
|--------|---------|-------------|
| $f(x)$ | $= x^2$ | for $x > 0$ |
| | $= 0$ | for $x = 0$ |
| | $= -x$ | for $x < 0$ |
- It is required to find the value of the function $y = f(x)$ for different values of x at different stages of a code
 - If the concept of function in the coding language is not known, then the programmer has to write the code for finding the value of $f(x)$ repeatedly as and when required
 - This unnecessarily increases the length of the code, and requirement of program memory
 - Instead, one writes, only once, a function $f(x)$ to find out its value and calls the function as and when required for different values of x
 - Whenever, the function is called, the address of the next instruction is pushed into the stack and the control is transferred to the first instruction of the function.
 - The function is thus executed, till the last statement, to find out its value
 - Normally the last statement of the function is a Return statement
 - This statement returns the value to the calling program and starts execution from the next instruction from where it was called (the address of this instruction is popped from the stack)

Functions

- A function is a way of packaging a group of statements (which perform a given task) for later execution
- Very often these functions are called repeatedly (or many times)
- The function is given a name that becomes a shorthand to describe the process
- A function is defined, in Python by ‘def’ statement as given below

```
def fname(a,b,c):  
    statement 1  
    statement 2  
    .....  
    statement n
```

- Function definitions are ‘digested’ for future use
- Function must have to be defined before it is used

Functions

- `def fname (a,b,c,...,z):`
 - statement1
 - statement2
 -
 - `return (v)`
- fname is the name of the function and a,b,c, ... ,z are the arguments
- statement1, 2, ... upto `return(v)` are in the body of the function which are uniformly indented
- Whenever ‘return’ statement is executed, the control is transferred to the calling program with a value of the return parameter
- Return value is obtained by calling the function as `val = fname (a,b,c,...,z)`
- Return value may be ignored by calling the function as `fname (a,b,c,...,z)`
- If there is no ‘return’ statement in the function definition, the full body of the function is executed before returning to the calling program
- The function returns ‘NONE’ to the calling program, if there is no ‘return’ statement

Example of a function

- # power(x,n) is a function to compute the value of x to the power n
- def power (x,n):

```
    ans = 1
    for i in range(0,n):
        ans = ans * x
    return(ans)
```
- # It is suggested to have two line gap after the last statement of a function
- a = 2
- b = 10
- print(power(a,b)) # In function 'power', x becomes 2 and n becomes 10
- # It is as good as having assignment statements in the function

```
x = 2
n = 10
```
- So same rules apply for mutable and immutable values

Passing values to a function

- If the passing value is **Immutable** (viz. numbers or strings)
 - For $x = y$, value in x and value in y are disjoint
 - Change in the value of x does not affect the value of y and vice versa
 - So these values will not be affected at calling point
- If the passing value is **Mutable** (like list)
 - For $x = y$, same value is shared by both x and y
 - Change in the value of x does affect the value of y
 - So these values will be affected at calling point

Passing mutable and immutable values to a function

- ```
• def update (lst, pos, val): # Mutable: lst Immutable: val , pos
 if pos >=0 and pos < len(lst):
 lst[pos] = val
 return(True)
 else:
 val = val *2 # changing the immutable values val, pos
 pos = pos * pos
 print ('val = ', val, 'pos = ', pos) => val = -8, pos = 36
 return (False)
```

- $(l, p, v) = ([1,3,5], 2, -4)$
  - $\text{stat} = \text{update}(l, p, v)$
  - $\text{print}(l, p, v, \text{stat})$   $\Rightarrow [1,3,-4], 2, -4, \text{True}$

- $(l, p, v) = ([1,3,5], 6, -4)$
  - $stat = update(l,p,v)$
  - $print(l,p,v,stat)$   $\Rightarrow [1,3,5], 6, -4, \text{False}$

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 6: 01/11/2021

Dr. Debranjan Sarkar

# Scope of names or variables

- Names within a function are local to it i.e. have local scope
- Example:
- `def func(x,y,z):`
  - `a = 5`
  - `v = x * y * z`
  - `return(v)`
- # Names inside a function are separate from names outside
- `(a,p,q,r) = (15,2,3,4)`
- `val = func(p,q,r)`
- `print('a = ',a)`                            $\Rightarrow$            `a = 15`

# Passing arguments by names

- # power(x,n) is a function to compute the value of x to the power n
- def power (x,n):
  - ans = 1
  - for i in range(0,n):
    - ans = ans \* x
  - return(ans)
- print(power(2,10)) => prints 1024
- print(power(10,2)) => prints 100
- print(power(n = 10, x = 2)) # If we forget the sequence of arguments

# Default arguments

- Python allows default values of arguments

```
def func(x, y=10):
```

.....

- Here at the definition stage of the function ‘func’, it is stated that the function has two arguments, out of which the default value for the second argument is taken as 10.
- At the time of calling the function, if the second argument is not provided, it is, by default, taken as 10.
- If the second value is provided, that value is taken as the value of the second argument
- Default value must be provided at the time of function definition

# Default arguments

- Default value has to be static and cannot be computed
- `def sort(a, p = 0, l = len(a)):` is not allowed as `len(a)` is not static
- The function `int` has actually two arguments with the second argument as base of the number, with default value of 10
  - `def int(s,b=10):` # convert a string to a decimal number (by default)  
.....
  - `int("95")` gives the result 95 (as it is a valid decimal number)
  - `int("B5")` gives rise to error (as it is not a valid decimal number)
  - `int("B5",16)` does not give any error as it takes B5 as a number with base 16
  - `int("B5",16)` gives the hexadecimal equivalent of B5 = 181

# Default arguments

- Default values must come at the end
- Default values are identified by the position
- Order of the arguments is important
- Example:
  - `def func(w,x,y,z):` # okay, no default values
  - `def func(w=10,x,y,z):` # wrong
  - `def func(w,x,y=10,z=18):` # okay
  - `func(5,8)` => `w = 5, x = 8, y = 10, z = 18`
  - `func(5,8,15)` => `w = 5, x = 8, y = 15, z = 18`
  - `func(5,8,15,38)` => `w = 5, x = 8, y = 15, z = 38`
- Not possible to use default value of y and given value of z by using 3 arguments

# Defining a function

- def is used to define a function
- A function can be defined conditionally

if condition:

```
def func(x,y,z):
```

.....

else:

```
def func(x,y,z):
```

.....

- Function can also be redefined
- Function can be assigned a new name

```
def func1(x,y,z):
```

.....

```
func2 = func1
```

- Now func2 is another name for func1
- This is required to pass a function as argument of another function (just like a variable)

# Function as an argument of another function

- Let us define a function ‘apply’ to apply, n number of times, a function ‘func’ with argument x
- ```
def apply(func,x,n):  
    ret_val = x  
    for i in range(n):  
        ret_val = func(ret_val)  
    return(ret_val)
```
- ```
def square(x):
 return(x * x)
```
- `apply(square, 9, 3)`      In ‘apply’, func = square; x = 9; n = 3
- $\text{result} = \text{square}(\text{square}(\text{square}(9))) = (81 \times 81) \times (81 \times 81) = 43046721$

# Recursion: Recursive function

- A function, which calls itself, is called a recursive function
- It is based on inductive definition
- Multiplication is basically repetitive addition
  - $m \times 1 = m$  Base case
  - $m \times n = m + m \times (n-1)$  Inductive step
  - As  $m \times n = m + m + m + \dots + m$  (n times)
- Factorial
  - $0! = 1$  Base case
  - $n! = n * (n-1)!$  Inductive step
  - As  $n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$
- Fibonacci series (1,1,2,3,5,8,13,21,.....)
  - $\text{Fib}(1) = \text{Fib}(2) = 1$
  - $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 7: 02/11/2021

Dr. Debranjan Sarkar

# Recursive function: Factorial and Fibonacci Number

- def fact(n): # Find out the factorial of n
  - if n <= 0:
    - return(1)
  - else:
    - val = n \* fact(n-1)
    - return(val)
- =====
- def Fib(n): # Find out the n<sup>th</sup> Fibonacci Number
  - if n == 0 or n == 1:
    - return (1)
  - else:
    - return(Fib(n-1) + Fib(n-2))

# Recursive function: Limit on recursion depth

- Python has a recursion limit of about 1000
  - When this exceeds it gives the following error:
  - RecursionError: maximum recursion depth exceeded in comparison
- 
- This limit may be extended manually as follows:
  - `import sys`
  - `sys.setrecursionlimit(5000)`

# Inductive definitions for lists

- Lists can be decomposed as:
  - First (or last) element
  - Remaining list without the first (or last) element
- List functions can be defined inductively as:
  - A base case: an empty list or a list containing only one element
  - Inductive step: function(lst) in terms of smaller sublist of lst
- To find out ‘Sum of a list of numbers’ inductively:
- Let the given list be [x<sub>1</sub>, x<sub>2</sub>, x<sub>3</sub>, ..., x<sub>n</sub>]

Sum [] = 0

# Base case

Sum[x<sub>1</sub>, x<sub>2</sub>, x<sub>3</sub>, ..., x<sub>n</sub>] = x<sub>1</sub> + sum[x<sub>2</sub>, x<sub>3</sub>, ..., x<sub>n</sub>]

# Inductive step

def sum\_list(lst):

    if lst == []:

        return(0)

    else:

        return (lst[0] + sum\_list(lst[1:])))

# Input/ Output

- It is required to interact with the users to receive input from them and to show them the results
- Standard input/output
  - Receive input from the keyboard
  - Display output to screen
- Reading from keyboard
  - `s = input()` # Read a line of input (no prompt) and assign it to s
  - `s = input('Enter your name: ')` # Prompt the user to type his/her name
  - Add space or new line (`\n`) to make the prompt more readable
  - `s = input('Enter a number:\n')`
  - s is a string
  - To type-convert it to integer, we use `int(s)`

# Standard output: Display on the screen

- `print(x, y, z)` # Prints values of names separated by spaces
- In Python 3, brackets are mandatory (optional in Python 2)
- `print('Hello, How are you?')` # Prints a message
- `print('x = ', x, 'y = ', y)` # Intersperse messages and values of names
- By default, print statement appends a \n (new line) character at the end
- This may be changed as follows:
  - `print('Hi', end = ' ')` # space at the end
  - `print('Hi', end = '$$ ')` # will print Hi\$\$ without new line

# Standard output: Display on the screen

- Items at the display due to a print statement are separated by space, by default (each comma generates a space). Example:

- `(x,y) = (5,9)` # Simultaneous assignment

- `print('x =',x,'y =',y)` # will display x = 5 y = 9

- This may be changed as follows:

```
print('x =',x,'y =',y,sep=':')
```

# will display x =:5:y =:9

```
(d,m,y) = (15,8,1947)
```

```
print(d,m,y,sep='/')
```

# will display 15/8/1947

```
print(d,m,y,sep=':')
```

# will display 15.8.1947

# Files

- Standard input/output is convenient when the volume of data is small
- Large volume of data normally resides in disk as files
- Files are to be read and written in lieu of standard input and output
- Disk R/W is much slow compared to memory R/W



- So a buffer (temporary storage) is required between the memory and the disk
- Data transfer takes place in terms of blocks (may be 2 Kbytes)
- For R/W of disk, firstly, we have to set up this buffer for the file
- This is done by opening a file which creates a file handle for getting access to the buffer
- Read/write operations are always through this file handle
- After R/W operation, the file is required to be closed which
  - Writes out (flushes) the buffer into the disk, and
  - Disconnects the file handle

# Files

- Opening a file:
- `fh = open("e:/file.txt", "r")` # file name: "file.dat" , fh = file handle
- 2<sup>nd</sup> argument is mode for opening
  - "r" : for read only     "w" : for write (overwrite) to a file     "a" : for append
- Reading from a file:
  - `s = fh.read()` # reads entire file (upto EOF) into s as a single string
  - `s = fh.readline()` # reads one line (upto and including \n) into s, unlike input()
  - `los = fh.readlines()` # reads entire file as list of strings los  
# each string is one line ending with \n
- Reading is a sequential operation
- When the file is opened, it points to position 0
- Each successive readline() moves forward till a \n is obtained

# Files

- `fh.seek(n)` moves pointer to position n
- `fh.seek(0)` moves pointer to the beginning of the file (position 0)
- `pan_number = fh.read(10)` # reads a fixed number (10) of characters
- For incremental reading of a file, it is required to know whether EOF has reached. At EOF,
  - `fh.read()` returns an empty string ""
  - `fh.readline()` also returns empty string
- Writing to a file
  - `n = fh.write(s)` writes string s to file & returns the no. of characters written
  - It is useful to know the value of n, as sometimes the statement may not write the desired number of characters due to some reason (e.g. disk is full)
  - `fh.writelines(los)` writes a list of strings (los) into the file
  - If the last character of every string in the list los is \n, it is as good as writing lines

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 8: 03/11/2021

Dr. Debranjan Sarkar

# Files

- **fh.close()** closes the file
  - Flushes the output buffer (all pending writes copied to disk)
  - Decouples file handle
- **fh.flush()** manually flushes the file
  - It forces all pending writes to copy to disk, without closing the file
- Processing file line by line

```
los = fh.readlines() # 1st method
```

```
for l in los:
```

```
.....
```

```
=====
```

```
for l in fh.readlines(): # 2nd method
```

```
.....
```

# Copying a File

- `infile = open("in-data.txt", "r")`
- `outfile = open("out-data.txt", "w")`
- `for line in infile.readlines():`
  - `outfile.write(line)`
- `infile.close()`
- `outfile.close()`
- =====
- `infile = open("in-data.txt", "r")`
- `outfile = open("out-data.txt", "w")`
- `lst = infile.readlines()`
- `outfile.writelines(lst)`
- `infile.close()`
- `outfile.close()`

# String manipulation

- String processing functions are useful to analyse and transform the contents of a text file
- `t = s[ :, -1]` # To get rid off trailing \n
- `t = s.rstrip()` # Removes (or strips) trailing whitespace (space, horizontal tab, linefeed, return, formfeed, and vertical tab)
- `t = s.lstrip()` # Removes leading whitespace including tabs
- `t = s.strip()` # Removes both leading and trailing whitespaces
- `n = s.find(p)` # Finds out the existence of pattern p in string s  
# returns the position of 1<sup>st</sup> occurrence of pattern  
# returns -1, if pattern not found
- `n = s.find(p,l,r)` # Finds out the existence of pattern p in the slice s(l:r)
- `n = s.index(p,l,r)` # Similar to find, but  
# returns “ValueError: substring not found” if pattern not found

# String manipulation

- `t = s.replace(old, new) #` t contains a copy of s with all occurrences of old string in s replaced by new string
- `t = s.replace(old, new, n) #` same as above except that it replaces at most n occurrences
- `print('ABABA'.replace('ABA','CC'))` # will print CCBA and not CCCC
- **Splitting a string:**
- `s = "1,2,3,4"` # s is a string with comma-separated values (CSV)
- `t = s.split("")` # t becomes ['1', '2', '3', '4'] a list of strings (split into chunks between commas)
- `t = s.split("", 2)` # t becomes ['1', '2', '3,4'] because it has split into at most 2 chunks
- `s = "4::5::6"` # s is a string with values separated by the string "::"
- `t = s.split("::")` # t becomes ['4', '5', '6']

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 9: 08/11/2021

Dr. Debranjan Sarkar

# String manipulation

- Joining a list of strings into a string separated by a given separator
- `s = ['1', '2', '3', '4']` # s is a list of strings
- `t = '-'.join(s)` # t becomes '1-2-3-4' (joined by dash (-) as separator)
- `"heritage".capitalize()` # will return "Heritage" (1<sup>st</sup> letter in uppercase)
- `"HERITAGE".lower()` # will return "heritage" (uppercase to lower)
- `"heritage".upper()` # will return "HERITAGE" (lowercase to upper)
- `"heritage institute of technology".title()` # will return "Heritage Institute Of Technology"
- `"HeriTage".swapcase()` # will return "hERItAGE"  
# uppercase to lower case and vice versa (swap cases)  
**so on so forth .....**

# String manipulation (Resizing)

- `s = 'Heritage'`
- `s.center(20)` # will return ' Heritage ' (total 20 characters)
- `s.center(20, '*')` # will return '\*\*\*\*\*Heritage\*\*\*\*\*'
- `s.ljust(20, '*')` # will return 'Heritage\*\*\*\*\*'
- `s.rjust(20, '*')` # will return '\*\*\*\*\* Heritage'
- `t.isalpha()` # will return True if string t contains only alphabets
- `t.isnumeric()` # will return True if string t contains only numbers  
so on and so forth .....

# Formatted Printing: String format() method

- (nb, ng) = (39, 41)
- print("No. of boys: {0}, No. of girls: {1}".format(nb,ng))  
would print => No. of boys: 39, No. of girls: 41
- print("No. of boys: {0}, No. of girls: {1}".format(47,51))  
would print => No. of boys: 47, No. of girls: 51
- print("No. of girls: {1}, No. of boys: {0}".format(47,51))  
would print => No. of girls: 51, No. of boys: 47
- print("No. of boys: {nb}, No. of girls: {ng}".format(ng = 51, nb = 47))  
would print => No. of boys: 47, No. of girls: 51
- print("No. of boys: {0:5d}, No. of girls: {1:6d}".format(47, 51))  
would print => No. of boys: 47, No. of girls: 51
- print("No. of students: {0:5d}, Average age: {1:6.2f}".format(47, 20.367))  
would print => No. of students: 47, Average age: 20.37  
Left justify, add leading zeros etc etc.....

# Exception handling

- Two types of errors we generally encounter, while running a code:
  - Compilation Error: Example: `print(5;3)` => SyntaxError: invalid syntax
  - Execution error or Run-time error => No error during compilation but error at run time
- When there is a run-time error, an error message is displayed as follows and the program is aborted

`TypeError` : brief diagnostic information

- Different sources of run-time error:
  - `x = 5 y = 0 z = x /y` => Division by zero not possible  
=> `ZeroDivisionError`: division by zero
  - `x = int('A5')` => string 'A5' is an invalid decimal number  
=> `ValueError`: invalid literal for int() with base 10: 'A5'
  - `y = x + 10` => x undefined, so y cannot be computed  
=> `NameError`: name 'x' is not defined

Contd....

# Exception handling

- Different sources of run-time error:
  - $x = lst[i]$  where  $i$  is not a valid index for the list  
=> **IndexError: list assignment index out of range**
  - Try to read from a file which does not exist  
and so on and so forth .....
- Some errors can be predicted or anticipated others are unexpected
- Predictable errors are called exceptions
- For exceptional situations, we should have some contingency plan
- In computer, this is called exception handling

# Exception handling

- ‘File not found’ exception might be due to typing mistake of user in providing the file name
- It can be handled by requesting the user to retype the file name
- ‘IndexError: List index out of bounds’ exception may be due to some mistake in the codes such that it tries to access a list item which is not there
- It can be handled by providing the value of the index to the user for easy debugging of the code
- Python provides some mechanism for handling such exceptional situations, without aborting the program
- If the exception is not handled, it would abort the execution of the program

# Exception handling

- try:
  - .....
  - .....
- except IndexError: # when index error occurs
  - .....
  - .....
- except (NameError, KeyError): # when there is a name error or key error
  - .....
  - .....
- except: # otherwise (for all other exceptions)
  - .....
  - .....
- else:
  - .....
  - .....# if no errors were raised (try block is  
# executed successfully)  
# (similar to else in for or while)

# Exception handling

- All these blocks are executed in sequence
- If both IndexError and NameError are generated, only IndexError code will be executed and not the NameError code
- Similarly for both ZeroDivisionError and NameError, only NameError code will be executed and not the ZeroDivisionError

# Reading from keyboard: Exception handling

- `while(True):`  
    `try:`  
        `s = input('Please enter a decimal number: ')`  
        `num = int(s)`  
    `except ValueError:`  
        `print('Please enter a valid number: ')`  
    `else:`  
        `break`

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 10: 15/11/2021

Dr. Debranjan Sarkar

# Exception handling in nested functions

- Suppose a function f calls a function g
- The function g calls another function h
- Suppose there is an exception while executing the function h
- This exception (say IndexError) is not handled in function h
- So the function h will abort (not the program in totality)
- So the IndexError goes back to the function g at the calling point
- As if function g has generated the IndexError
- If function g does not have a ‘try’ block, this error is transferred to the function f at the calling point
- If function f does not handle this error it is transferred to the main program calling the function f at the calling point
- If it is not handled here also, the program aborts
- The exception can be handled in any of the functions

# Nested Functions

- def h():
  - .....
  - .....
  - Exception # No try block in function h()
  - .....
- def g():
  - .....
  - .....
  - x = h()
    - ..... # No try block in function g()
- def f():
  - .....
  - .....
  - y = g() # No try block in function f()
  - .....
- .....
- z = f() # If there is no try block in main program, the program is aborted
- .....

# Module 2

# Python Data Structures

List, Tuple, Set, Dictionary,  
Stack and Queue

# What is data structure?

- Data Structures deal with the organization and storage of data in the memory, while a program is processing the data
- Data Structure explains the relationship between the data and the operations that can be performed on the data
- Some of the useful data structures are:
  - Arrays and lists: sequences of values
  - Dictionaries: key – value pairs (useful for maintaining various types of information)
  - Python has a built-in implementation of sets
  - Stacks are useful to keep track of recursive computation
  - Queues are useful for breadth-first exploration

# Tuples (Immutable)

- We know Simultaneous assignments `(a,b,c) = ('Sourav', 45, 'True')`
- A sequence of values within round brackets is called a Tuple
  - Pairs, triples, quadruples, quintuples, sextuples, .... , n-tuples
- Python can assign a tuple of values to a single name. Example:
  - `student_name = (first_name, middle_name, surname)`
  - `point_2d = (x_coor, y_coor)`
  - `date = (26,12,2019)`
  - `empty_tuple = ()` # No element within parentheses
  - `one_element_tuple = (12,)` # A trailing comma is a must to distinguish it from integer within ()
- These are not lists but tuples
- We can extract one element or a slice from a tuple. Example:
  - `student_surname = student_name[2]`
  - `month_year = date[1:]` (12,2019)
- Tuples are basically immutable version of lists
  - `date[1] = 11` will give rise to error

# Sets (Mutable)

- A set is like a list with curly brackets, where
  - Duplicates are automatically removed, and
  - No inherent order to the elements
- `s = {'a', 'b', 'c', 'a', 'e'}`
- `print(s)` will print only `{'a', 'b', 'c', 'e'}` # 'a' only once
- To create an empty set, do not use `s = {}` as it indicates an empty dictionary
- To create an empty set, we use `s = set()` with no arguments
- Main advantage of using set (as opposed to a list)
  - Set has highly optimized method for checking the membership of an element
- Like list, set membership can be checked by the 'in' operator
- 'c' in s will return True and 'z' in s will return False

# Sets

- A list can be converted to a set by the set function.
- `s = set(['x', 'y', 'z', 'p', 'z'])` will return the set `{'x', 'y', 'z', 'p'}`
- `letters = set('school')` will return the set `{'o', 's', 'l', 'h', 'c'}`

## Set Operations

|                         |                                                           |
|-------------------------|-----------------------------------------------------------|
| • odd = {1, 3, 5, 7, 9} | prime = {2,3,5,7}                                         |
| • Union operation:      | <code>odd   prime</code> $\Rightarrow$ {1, 2, 3, 5, 7, 9} |
| • Intersection:         | <code>odd &amp; prime</code> $\Rightarrow$ {3, 5, 7}      |
| • Set difference:       | <code>odd – prime</code> $\Rightarrow$ {1, 9}             |
| • Exclusive OR:         | <code>odd ^ prime</code> $\Rightarrow$ {1,2,9}            |

# Sets

- Two major restrictions
  - Set does not maintain the elements in any particular order
  - Values of only immutable types can be the elements of a Python set
- $i = \{2, 6, 9\}$        $f = \{98.4, 3.14\}$        $s = \{'a', 'n', 'b'\}$  are all valid sets
- A set of tuples is also valid
- A set of list or a set of sets are invalid as lists and sets are mutable

# Frozen Sets (Immutable)

- The ‘frozenset’ is an immutable form of ‘set’ type
- It is legal to have both mutable and immutable datatypes in a frozenset
- `s = frozenset({3, 5, 7, 5})`
- `t = frozenset({98.4, 3.14})`
- `u = {s,t}` u is a set of two elements, which are frozen sets
- `print(u)` # will display {frozenset({3, 5, 7}), frozenset({98.4, 3.14})}

# Dictionaries

- Dictionary is another built-in Python type
- A dictionary is an indexed data structure (like list)
- However, in dictionary, the indices are not positions, but values
- Any immutable type (e.g. string, numbers) can be used as an index
- An empty pair of curly brackets creates a dictionary             $d = \{\}$
- New elements are placed in the dictionary by using an index in an assignment
- $d['name'] = 'Dipak Poddar'$
- $d['age'] = 20$
- $d['sex'] = 'M'$
- The index is called a key and the element associated with the key is called a value
- A dictionary can be initialized, at the time it is created, as follows:
- $d = \{'name': 'Dipak Poddar', 'age': 20, 'sex': 'M'\}$

# Some common operations for a dictionary

- `len(d)`      =>      number of elements in dictionary d
- `d[k]`          =>      item in dictionary d with key k
- `d[k]=v`        =>      set item in d with key k to value v
- `del d[k]`      =>      delete item with key k from dictionary d
- `d.clear()`     =>      removes all the items from the dictionary d
- `d.items()`     =>      return a list of (key, value) pairs  
                          =>      [ ('name', 'Dipak Poddar'), ('age', 20), ('sex', 'M') ]
- `d.keys()`       =>      return a list of keys in d  
                          =>      ['name', 'age', 'sex']
- `d.values()`     =>      return a list of values in d  
                          =>      ['Dipak Poddar', 20, 'M']
- `d.get(k)`       =>      item in dictionary d with key k (same as `d[k]`)  
                          =>      'None', if key is invalid
- `d.get(k, v)`     =>      returns `d[k]`, if key is valid, otherwise returns value v

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 11: 16/11/2021

Dr. Debranjan Sarkar

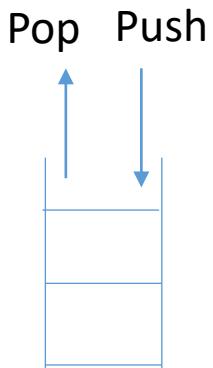
# Positive use of exception in dictionaries

- We have a dictionary:
- `scores = {'Kohli': [175,30], 'Dhoni': [69,45]}`
- We have two keys (strings) 'Kohli' and 'Dhoni', to which are associated their scores (a list)
- In a particular match we have score s for batsman b
- It is required to update the dictionary with this information
- Two situation may arise
  - Batsman b already exists in dictionary  
`scores[b].append(s)`      ->      append score s for batsman b
  - Batsman b is new  
`scores[b] = [s]`      ->      create fresh entry

# Positive use of exception in dictionaries

- Style 1: Traditional approach:
- if b in scores.keys():
  - scores[b].append(s)
- else:
  - scores[b] = [s]
- =====
- Style2: Using Exception Handling:
- try:
  - scores[b].append(s)
- except KeyError:
  - scores[b] = [s]

# Stack



- In a list, we can freely insert or delete values at any position of the list
- Whereas, stack is a last-in first-out (LIFO) list
- New element is added only from one end of the list and the element to be removed is from the same end
- Two operations are possible in a stack:
  - Push operation: `push(s,x)` => adds value x into the stack s
  - Pop operation: `pop(s)` => returns the most recently added element in stack s
- A list can be used as a stack
  - An element is added (pushed) into the right of the list
  - An element is popped out of the list from the right
- `push(s,x)` is equivalent to `s.append(x)`
- `pop(s)` is equivalent to `s.pop()`
- `s.pop()` is a built-in function to return the last element of the list
- Stacks are required for recursive function calls

# Queue



- Unlike stack, queue is a first-in first-out (FIFO) list
- New element is added from one end of the list and the element to be removed is from the other end
- `addq(q,x)` adds the value  $x$  to the queue from the rear
- `removeq(q)` removes element from the head of the queue
- We may assume a queue as a list, with the front of the queue towards the right of the list and the rear of the queue at the left of the list
- `lst.insert(j,x)` inserts the value  $x$  before the  $j^{\text{th}}$  position of the list `lst`
- So `addq(q,x)` is equivalent to `q.insert(0,x)`, assuming  $0^{\text{th}}$  position at the left
- `removeq(q)` is equivalent to `q.pop()`

# Module 1

# Data Organization

Files and Exceptions

Classes, objects, inheritances

Object Oriented Programming in Python

# Classes, objects, inheritances

# Data structure revisited

- Data structure is an organization of information whose behaviour is defined through interface
- Interface is basically an allowed set of operations
- For every datatype, there is an allowed set of operations
- Example:
  - For stack                          => push() and pop()
  - For queue                          => addq() and removeq()
- We have seen how data structures (stacks and queues) can be implemented using list type in Python

# Built-in datatypes in Python

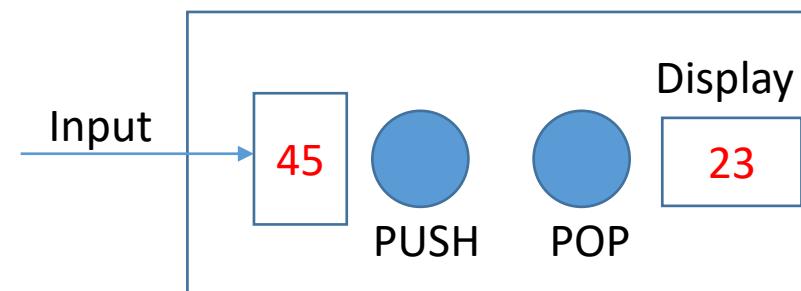
- We already know that certain operations are allowed for a particular built-in data type in python
- For example,
- `lst = []` # `lst` is defined as a list (a built-in data type)
- This implies that operations such as `append`, `extend` etc. are permitted
- So, `lst.append()`, `lst.extend()` are allowed
- `lst.keys()` is not permitted as `keys` is a valid operation for dictionary but not list
- Similarly, `dic = {}` # `dic` is defined as a dictionary
- Here, ‘values’ operation is allowed but not the operation ‘append’
- `dic.values()` is allowed but not `dic.append()`
- Besides built-in data types in Python, we can create our own datatypes

# Abstract datatypes

- We define new abstract datatype in terms of the operations allowed
- Implementation details are not of interest / not referred
- Implementation can be optimized without affecting functionality
- Abstract definition of:
  - Stack:  $(s.push(v)).pop() == v$
  - Queue (initially q is empty):  $((q.addq(u)).addq(v)).removeq() == u$
- Abstract datatype can be viewed as a black box with :

Public Interface

Private implementation



First, we POP and get 23  
Then we PUSH value 45

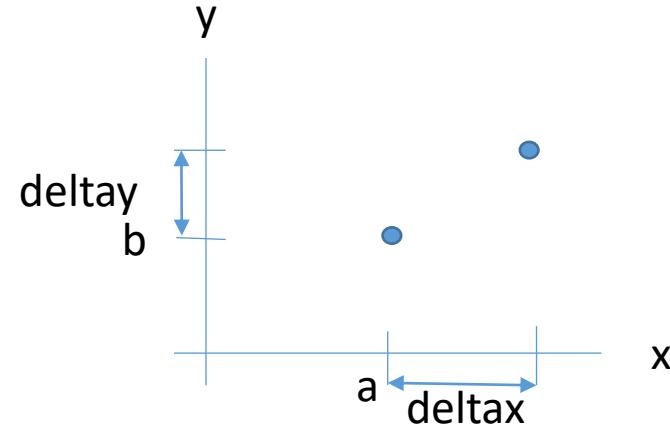
# Object Oriented Programming (OOP)

- A style of programming, where one of the main feature is to define a new data type in terms of Public Interface (operations allowed on that type of data)
- Separately, an implementation, which should ideally be private, is provided
- In Python, full notion of privacy is not maintained
- Two important concepts of OOP are Classes and Objects

# Classes

- Class is a template (or blueprint) of data type (like a function definition)
- It carries information about
  - How data is stored
  - What are the operations allowed and how public functions manipulate data
- Example:
- class Point: # defining a class Point on a 2-D plane with coordinates (x,y)

```
def __init__(self, a, b):
 self.x = a
 self.y = b
```



```
def translate(self, deltax, deltay):
 self.x += deltax
 self.y += deltay
```

# translate the point by (deltax, deltay)  
# equivalent to self.x = self.x + deltax  
# equivalent to self.y = self.y + deltay

# Objects

- Once we have a template for a data type, we can construct many instances of it
- When we have a blueprint of a stack, we can have many instances of a stack
- These instances of the class are called objects
- Logically, each of the objects has a copy of the function associated with the class
- In classical programming, we have one function `push(s, v)`, where `s` is a parameter to provide the information about stack
- In OOP, there is a ‘push’ associated with each of the stacks (say `s1, s2, s3,...`)
- So, in OOP we say `s1.push(v1)`      in place of `push(s1, v1)` as in classical programming
- If we define a function ‘append’ to append a value to a list, it is expected to have two arguments as follows:  
`append(lst1, val1)`
- In OOP, `lst` is an object and ‘append’ is a function to update the object. So OOP notation is as follows:  
`lst1.append(val1)`

# Classes and Objects: Translation of a point

- Let p be an instance of the class Point. So p is an object of class Point
- Let the point p be initialized as  $p = \text{Point}(4,3)$
- For initialization of the point p, the `__init__` function under the class Point is called with  $a = 4$  and  $b = 3$
- Here, 'self' means the point p itself
- So,  $p.x = 4$  and  $p.y = 3$  which indicate the point  $(4,3)$  in 2-D plane
- $p.translate(1,3)$  calls the function with  $\text{deltax} = 1$  and  $\text{deltay} = 2$  and 'self' as the point p itself
- After this,  $p.x = p.x + \text{deltax} \Rightarrow p.x = 4 + 1 = 5$
- and,  $p.y = p.y + \text{deltay} \Rightarrow p.y = 3 + 3 = 6$

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 12: 18/11/2021

Dr. Debranjan Sarkar

# Classes and Objects: Distance of a point from origin

- class Point: # defining a class with name Point on a 2-D plane with coordinates (x,y)

```
def __init__(self, a, b):
 self.x = a
 self.y = b

def distance_origin(self):
 distsq = self.x * self.x + self.y * self.y
 return(sqrt(distsq))
```

# Find the distance of the point from the origin  
# from Pythagoras Theorem  
# requires statement 'from math import \*'

- Let p be an instance of the class Point. So p is an object of class Point
- Let the point p be initialized as p = Point(3,4)
- Then the `__init__` function under the class Point is called with a = 3 and b = 4
- Here, 'self' means the point p itself
- So, p.x = 3 and p.y = 4 which indicate the point (3, 4) in 2-D coordinate system
- `d = p.distance_origin()` calls the function with 'self' as the point p itself
- The value 5 is returned for d as the distance of the point (3,4) from the origin

# Classes and Objects: Point in Polar coordinates

## Coordinates of a point in Cartesian Coordinates (x,y)

## Coordinates of the point in Polar Coordinates ( $r, \theta$ )

$$r = \sqrt{x^*x + y^*y}$$

$$\tan \theta = y / x$$

```
class Point: # defining a class Point on a 2-D plane with coordinates (x,y)
```

```
def init (self, a, b):
```

```
self.r = sqrt(a * a + b * b) # we work in polar coordinates (r, θ)
```

if a == 0:

```
self.theta = 3.141519 / 2 # if a =0, the angle is pi /2
```

else:

```
self.theta = atan(b/a) # Otherwise, the angle is
tan inverse of (b/a)
```

# Classes and Objects: Point in Polar coordinates

# To find the distance of the point from the origin (Polar coordinates system is better)

```
def distance_origin(self):
 return(self.r) # self.r is basically the distance from the origin
```

# To shift or translate the point by (deltax, deltay) (Cartesian coordinates system is better)

```
def translate(self, deltax, deltay):
 self.x = self.r * cos(self.theta) # convert polar to Cartesian coordinates system
 self.y = self.r * sin(self.theta)
 self.x += deltax # Translate
 self.y += deltay
 self.r = sqrt(self.x * self.x + self.y * self.y) # convert Cartesian to Polar coordinates system
 if self.x == 0:
 self.theta = 3.141519 / 2
 else:
 self.theta = atan(self.y/self.x)
```

# Classes and Objects

- In the previous example, we have shown that the Private implementation has changed but the functionality of the Public interface remains same
- Default arguments may be provided while defining a function
- class Point:

```
def __init__(self, a=0, b=0):
 self.x = a # p1 = Point(5,3) => point p1 is at (5,3)
 self.y = b # p2 = Point() => point p2 is at (0,0)
```

# Special Functions

- `__init__` is a special function we have already encountered
  - The `__init__` function is termed a **Constructor**
  - It is used to initialize a newly created instance of the class
  - The constructor is never directly invoked
  - Instead, it is called implicitly as part of the process of creating a new object
- `__str__` returns string representation of an object
  - To print out the value of an object, this function is implicitly invoked
  - `str(o)` is equivalent to `o.__str__()`
  - For the class `Point()`, the special function `__str__` may be defined as follows:
    - `def __str__(self):` `p = Point(5,7)`
    - `return('(' + str(self.x) + ', ' + str(self.y) + ')')` `print(p)`
    - `will display (5,7)`

# Special Functions

- The special function `__add__` is invoked implicitly by the `+` operator
  - `p1 + p2` invokes the function `p1.__add__(p2)`
  - It is upto us how to define the function `__add__()`
  - `def __add__(self, p):  
 return(Point(self.x + p.x, self.y + p.y))`
  - `p1 = Point(5,1) p2 = Point(2,3) p3 = p1 + p2 => p3 = Point(7,4)`
- The special function `__mult__` is invoked implicitly by the `*` operator
  - `p1 * p2` invokes the function `p1.__mult__(p2)`
  - It is upto us how we define the function `__mult__()`
  - `def __mult__(self, p):  
 return(Point(self.x * p.x, self.y * p.y))`
  - `p1 = Point(5,1) p2 = Point(2,3) p3 = p1 * p2 => p3 = Point(10,3)`
- `__lt__`, `__gt__`, `__le__` are invoked implicitly by the `<`, `>`, `<=` operators respectively
- We define these functions as per our requirement
- There are several other special functions

# Classes and Objects (another example)

- # This is a very primitive code

- class BankAccount:

```
def __init__(self, a = 0):
 self.balance = a

def deposit(self, amount):
 self.balance += amount

def withdraw(self, amount):
 self.balance -= amount

def transfer(self, amount, destaccount):
 self.withdraw(amount)
 destaccount.deposit(amount)

def getbalance(self):
 return self.balance
```

# Classes and Objects (another example)

- ajain = BankAccount()
  - pbose = BankAccount(500)
  - ajain.deposit(10000)
  - ajain.transfer(1000, pbose)
  - pbose.withdraw(500)
  - print(ajain.getbalance())
  - print(pbose.getbalance())
- # ajain initial balance = 0
  - # pbose initial balance = 500
  - # ajain current balance = 10000
  - # ajain current balance = 9000,  
# pbose current balance = 1500
  - # pbose current balance = 1000
  - # Display ajain balance = 9000
  - # Display pbose balance = 1000

# Inheritance

- Deriving new classes from the existing classes such that the new classes inherit all the members of the existing classes, is called Inheritance
- The original class is called the **parent class** or **base class** or **super class**
- The newly created class is called the **child class** or **sub class** or **derived class**
- The syntax for inheritance

```
class childclass(parentclass):
```

- Example:
- Parent Class:      BankAccount
- Child class:        ChequeAccount

# Child Class ChequeAccount of Parent Class BankAccount

```
class ChequeAccount (BankAccount):
 def __init__ (self, initbal):
 BankAccount.__init__ (self, initbal)
 # New data field to create a dictionary for cheque information
 self.chequerecord = {}

 def processcheque (self, number, towhom, amount):
 self.withdraw(amount)
 # append cheque info into dictionary
 self.chequerecord[number] = (towhom, amount)

 def chequeinfo (self, number):
 return (self.chequerecord [number])
```

## Child Class ChequeAccount of Parent Class BankAccount

- # Create a Cheque Account with initial balance of Rs. 5000
- ca = ChequeAccount(5000)
- # Cheque No. 100 for Rs. 630/- is issued to Gas Company
- ca.processcheque (100, 'Gas Company', 630)
- # Cheque No. 101 for Rs. 1055/- is issued to Electric Supply
- ca.processcheque (101, 'Electric Supply', 1055)
- # Cheque No. 102 for Rs. 1000/- is issued to XYZ School
- ca.processcheque (102, 'XYZ School', 1000)
- # Functions of Parent Class can also be invoked by the object of Child Class
- ca.deposit(500)
- print(ca.getbalance()) # will display the balance in the account Rs. 2815
- print(ca.chequeinfo(101))# will display ('Electric Supply', 1055)

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python  
Lecture 13: 23/11/2021

Dr. Debranjan Sarkar

# More on Inheritance

- When inheritance is used to construct a new class, the code is inherited from the parent class and it need not be rewritten (software reuse)
- Software reuse is more important when same class is used as a parent class to multiple child classes
- Software reuse is more pronounced when the parent class developed by others is distributed as part of an existing library
- One needs to describe only the new methods and data values
- When a child class modifies or replaces the behaviour inherited from the parent class, this is called **overriding**
- To override a method, the child class redefines the function using the same name and arguments
- If the overriding function has to invoke a function in the parent class, the class name should be explicitly provided

# More on Inheritance

- Example of overriding:
  - # The child class ChequeAccount wants to print a message every time a withdrawal is made
- ```
class ChequeAccount (BankAccount):
```

.....

```
def withdraw(self, amount):
    print ("Withdrawing Rs. ", amount)
    BankAccount.withdraw(self.amount)
```

- Each class definition creates a new type. Example:
- `print (type(ajain))` => will display <class '__main__.BankAccount'>
- `print (type(BankAccount))` => will display <class 'type'>

More on Inheritance

- `print(isinstance(ca, ChequeAccount))` => will display True
- `print(isinstance(ca, BankAccount))` => will display True
- `print(isinstance(ajain, ChequeAccount))` => will display False
- `print(isinstance(ajain, BankAccount))` => will display True
- `print(issubclass(ChequeAccount, BankAccount))` => will display True
- `print(issubclass(BankAccount, ChequeAccount))` => will display False
- Class Variable with example:

```
class BankAccount:  
    accounttype = "Bank Account" # Defining a Class Variable accounttype  
    def __init__(self, a = 0):  
        self.balance = a
```

- Only one value is stored for a Class variables, which can be shared among all instances of the class
- The class variable can be printed using the dot notation as follows
- `print (ajain.accounttype)` will display "Bank Account"

Multiple Inheritance

- Deriving a child class from more than one parent classes is called multiple inheritance
- The syntax for single inheritance
- class childclass(parentclass):
- Syntax for multiple inheritance
- class childclass(parentclass1, parentclass2, ...):

Example of multiple inheritance:

```
class ParentClass1:  
    def parentfun1(self):  
        print("In Parent Class 1")
```

```
class ParentClass2:  
    def parentfun2(self):  
        print("In Parent Class 2")
```

```
class ChildClass(ParentClass1, ParentClass2):  
    def childfun(self):  
        print("In Child Class")
```

Multiple Inheritance

- Instances of the new class can use methods from either parent
- `cc = ChildClass()` # cc is an object of the class ChildClass
- `cc.childfun()` will display “In Child Class”
- `cc.parentfun1()` will display “In Parent Class 1”
- `cc.parentfun2()` will display “In Parent Class 2”
- Problems in Multiple Inheritance:
 - If the child class has a constructor, it overrides the constructor of the parent class and hence the parent class constructor is not available to the child class
 - However, writing constructor is very common to initialize the instance variables
 - This problem is discussed in the following example

Multiple Inheritance

- In the example the parent classes and the child class have their own constructors

```
class ParentClass1:  
    def __init__(self):  
        self.a = 'a'  
        print(self.a)
```

```
class ParentClass2:  
    def __init__(self):  
        self.b = 'b'  
        print(self.b)
```

```
class ChildClass(ParentClass1, ParentClass2):  
    def __init__(self):  
        self.c = 'c'  
        print(self.c)  
        super().__init__()
```

- Now, initialize an object o of class ChildClass
- o = ChildClass()
- This will display 'c' and 'a' in two successive lines ['b' will not be displayed]
- Had we defined ChildClass as class ChildClass(ParentClass2, ParentClass1):
o = ChildClass() would have displayed 'c' and 'b' in two successive lines [but not 'a']

Module 2: Manipulating strings

Regular expression in Python

- To find whether a fixed string is there in another string, we use the method ‘find’
- Example:
- `s = 'examination'`
- `print(s.find('in'))` will display 4 as the first occurrence of ‘in’ is found at position 4 in the string ‘examination’
- To find whether a variable string (containing variable characters and/or of variable length) is there in another string, we use a technique called **regular expression (or, regex)** notation
- The pattern we want to find in a string is to be defined as a regular expression

Regular expression in Python

- Defining patterns in Regular Expression
- Pattern containing 3 alphanumeric characters and starting with ‘a’
 - ‘a\w\w’ # \w means any alphanumeric character (a – z), (A – Z), (0-9)
- The regular expression (re) module offers a set of functions:
- **Example:**
 - import re # re module is to be imported
 - s = ‘translation’
 - pat = re.compile(‘a\w\w’) # compiling the regular expression to an object pat
 - print(pat.search(s)) => display <span=(2, 5), match='ans'>
=> 1st occurrence only
=> ‘None’ returned, if no matches found
 - print(pat.findall(s)) => display a list ['ans', 'ati']; empty list, if not found
 - print(pat.sub('yyy', s)) => display tryyyllyyyon
 - print(pat.split(s)) => display a list ['tr', 'I', 'on']

Match Object

- A match object is an object containing information about the search and the result
- ‘None’ will be returned instead of the match object, if no match is found
- Example:
 - import re # re module is to be imported
 - s = ‘examination’
 - mo = re.search(‘a\w\w’, s) # This will return a match object mo
 - print(mo) => display <re.Match object; span=(2, 5), match='ami’>
 - mo = re.search(‘b\w\w’, s) # No match
 - print(mo) => display ‘None’

Match Object

- The Match Object has properties and methods used to retrieve information about the search and the result
 - `.span()` returns a tuple containing the start and the end position
 - `.string` returns the string passed into the function
 - `.group()` returns the part of the string, where there was a match
- In the earlier Example:
 - `print(mo.span())` => display (2, 5)
 - `print(mo.string)` => display 'examination'
 - `print(mo.group())` => display 'ami'

Metacharacters

- Metacharacters are characters with special meaning

• <u>Character</u>	<u>Description</u>	<u>Example</u>
• [chars]	one character from range	"[a-m]"
• [^chars]	one character not from range	"[^f-m]"
• \	signals a special sequence	"\d"
• .	Any character except newline	"he..o"
• ^	starts with	"^hello"
• \$	ends with	"world\$"
• *	zero or more occurrences	"aix*"
• +	one or more occurrences	"aix+"
• ?	Optional (zero or one)	"([+-])?"
• {m}	exactly m no. of occurrences	"a{2}"
• {m,n}	From m to n no. of occurrences	"a{2,4}"
•	either or	"falls stays"
• (...)	group	

Special Sequences

- A backslash followed by one of the following characters

<u>Character</u>	<u>Description</u>	<u>Example</u>
• \A	if the specified characters are at the beginning of the string	'\AThe'
• \b	if the specified characters are at the beginning or at the end of a word	'\bain' 'ain\b'
• \B	if the specified characters are present, but not at the beginning or at the end of a word	'\Bain' 'ain\B'
• \d	if the string contains digits (numbers 0 – 9)	'\d'
• \D	if the string does not contain digits (numbers 0 – 9)	'\D'
• \s	if the string contains whitespace characters [1]	'\s'
• \S	if the string does not contain whitespace characters [1]	'\S'
• \w	if the string contains any word character [2]	'\w'
• \W	if the string does not contain any word character [2]	'\W'
• \Z	if the specified characters are at the end of the string	'Spain\Z'

• [1] Whitespace characters are space, tab, linefeed, return, formfeed, and vertical tab

• [2] A word character means any alphanumeric (a-z, A-Z, 0-9) or underscore character

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 14: 25/11/2021

Dr. Debranjan Sarkar

Sets in Regular Expression: characters within []

- Set Description
- [bsz] one of the specified characters (b, s, or z) present
- [a-q] any lower case character (between a and q) present
- [^bsz] any character except b, s, or z present
- [1356] any of the specified digits (1, 3, 5 or 6) present
- [0-9] any digit between 0 and 9 present
- [0-6][0-8] any 2-digit number between 00 and 68 present
- [a-zA-Z] any character between a and z or A and Z
- [+] any + character in the string
- [*] any * character in the string
- Note: In sets, +, *, ., |, (), \$, {} have no special meanings

Examples of Regular Expression

- Define regular expression for floating point number or numeric string
- Examples: 2 .72 -1.72 +1.72 1.72 etc...
- Optional sign -> optional series of integers -> optional dot -> optional series of integers
- At first thought: $[+-]?[0-9]^*\.\.[0-9]^*$ Or $[+-]?[d]^*\.\.[d]^*$
- However, it treats the following as valid floating point number
 - Only sign (- or +)
 - Only dot (.)
 - A sign with dot (+. or -.)
 - Null string ("")
- To overcome this problem, we use the following regular expression:
 $[+-]?[0-9]^*\.\.[0-9]+$ Or $[+-]?[d]^*\.\.[d]+$

Examples of Regular Expression

- Floating point number can also be shown as follows

2E8 .72E-07 -1.72e+4 +1.72e-02 1.72E-3

- Optional sign -> zero or more integers -> optional dot -> one or more integers -> optionally (e or E -> optional sign -> one or more integers)
- The corresponding regular expression is as follows:

$[+-]?[0-9]^*\.\.?([0-9]+([eE][+-]?[0-9]+))?$

- The entire exponent part is made optional by grouping it together
- To validate if a particular string holds a floating point number, rather than finding a floating point number within a text, the regex is as follows:

$^[-+]?[0-9]^*\.\.?([0-9]+)$$

$^[-+]?[0-9]^*\.\.?([0-9]+([eE][+-]?[0-9]+))?)?$

Functional Programming

Different Programming Paradigms

- There are three different paradigms, used to describe the mental model of a programmer
- (1) **Imperative paradigm** is the model in which instructions have the effect of making changes to memory
- The task of programming is to place statements in their proper sequence, so that by a large number of small transformations to memory, the desired result is eventually produced
- (2) The **Object-oriented paradigm** is built on the mechanism of defining classes, creating instances of those classes, and using inheritance etc.
- The Object-oriented paradigm refers to the model of computation built on top of these facilities, and not just the mechanics of defining classes
- (3) The **Functional Programming** paradigm views the task of computation as a process of transformation
- By performing a sequence of transformation on a value, the desired result is produced
- The Functional Programming model requires the ability to create functions, but the paradigm is much more than the simple use of this mechanism

Functional Programming

- In functional programming, the creation of functions is an important part of the paradigm
- The key characteristic of functional programming is that it creates new values by a process of transformation
- Generally, values are represented as lists, or dictionaries
- In imperative paradigm, complex values are produced by *modification* i.e. a large number of small changes to an existing data structure
- Small changes may often be accompanied by small errors which may produce only a minimal effect. Thus debugging imperative programs are difficult
- As functional programs create new values by *transformation* which are often more uniform and much simpler to write and debug
- Errors do occur but these tend to be larger and thus easier to find and eliminate
- The process of transformation can be subdivided into several common forms viz. mapping, Filtering and Reduction

Mapping

- A mapping is a one-to-one transformation, where each element in the source is converted to a new value
- The new value is gathered into a collection, leaving the original collection unchanged
- Built-in function `map(f, lst)` applies the function `f` to each element of list `lst`
- Output of `map()` produces a sequence, not a list in Python 3
- Function `list(map(f,lst))` is used to convert it to a list

```
def squares(x):  
    return(x * x)  
lst = [1, 2, 3, 4]  
lst1 = list(map(squares,lst))  
print(lst1)          # will display [1, 4, 9, 16]
```

- Output of `map` function may be used directly in a for loop

```
for i in list(map(f,lst)):      or      for i in map(f,lst):
```

Filtering

- Filter function is useful to filter out the elements of a sequence depending on the result of a function
- The format is `filter(fun, sequence)`
- The function `fun` returns either True or False
- The sequence represents a list, string or tuple
- The function `fun` is applied to every element of the sequence when it returns True, the element is extracted otherwise it is ignored
- Define a function `is_even` which returns True if its argument is even, otherwise it returns False

```
def is_even(x):  
    return(x % 2 == 0)
```

- We can use this function inside `filter()` to select a sub-list of even numbers from a given list
- `lst = [3, 29, 4, 78, 99, 100]`
- `lst1 = list(filter(is_even, lst))`
- `print(lst1)` will display [4, 78, 100]

Using map and filter function

- Using map and filter functions to find out the sum of squares of even numbers from a given list lst
- def squares (x):
 - return (x * x)
- def is_even(x):
 - return(x % 2 == 0)
- list(map(squares,(filter(is_even,lst)))) will give us the list containing the squares of the even numbers from the given list
- Now we can sum all the elements of the obtained sub-list

Reduction

- Reduction is the process of applying a binary operation (+, * etc.) to each member of a list in cumulative fashion
- The reduce() function reduces a sequence of elements to a single value by processing the elements according to the function supplied
- Let the given list [1,2,3,4,5,6] is to be reduced using + operation
- The result would be (((((1+2) + 3) + 4) + 5) + 6) or 21

```
import functools      # 'reduce' function is defined in "functools" module
def cumulative_sum(x,y):
    return(x+y)
a = [1,2,3,4,5,6]
print(functools.reduce(cumulative_sum,a)) will display 21
```

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Introduction to Python
Lecture 15: 30/11/2021

Dr. Debranjan Sarkar

Anonymous function or Lambda function

- A function without a name is called anonymous
- Anonymous functions are not defined using ‘def’
- They are defined using the keyword **lambda**
- Two ways that a function is used as an argument:
 - Pass the name of a previously defined function (as we have seen earlier)
 - To define a nameless function as an expression (lambda)
- Example:
- `lst = [1,2,3,4,5,6,7,8,9]`
- `print (list(map(lambda x: x * 2 + 1, lst)))` $\Rightarrow [3,5,7,9,11,13,15,17,19]$
- `print (list(filter(lambda x: x % 2 == 0, lst)))` $\Rightarrow [2,4,6,8]$
- `print (functools.reduce(lambda x,y : x + y, lst))` $\Rightarrow 45$
- Note that the function ‘filter’ has one argument (x) and returns a Boolean value (True or False).
- This type of one-argument function that returns a Boolean result is called a **predicate**

List comprehensions

- It is a simpler form of functional programming
 - In this form, lists can be characterized by a process as:

[expression1 for variable in list if expression2]

- Here, expression1 -> map

for variable in list -> generator

if expression2 -> filter

- Examples:

```
a = [1,2,3,4,5,6]
```

```
print([x*x for x in a if x > 2 and x < 5])
```

will display [9,16]

```
print([x*x for x in range(50) if x%2 == 0])
```

will display squares of even numbers < 50

```
def list_of_squares(a):
```

```
return [x*x for x in a]
```

will return [1,4,9,16,25,36]

```
print([a[i] for i in range(0,len(a)) if i%2 == 0])
```

will print every other element in the list [1,3,5]

`d = {2: 'HITK', 7: 'IU', 15: 'IEM', 22: 'NIT'}`

d is a dictionary

```
print([d[i] for i in d.keys() if i%2 == 0])
```

Example: Pythagorean triple

- (x,y,z) is a Pythagorean triple if $x^2 + y^2 = z^2$
- Find out $\{(x,y,z) \mid 1 \leq x,y,z \leq n \text{ and } x^2 + y^2 = z^2\}$
- `print([(x,y,z) for x in range(50) for y in range(50) for z in range(50) if x*x + y*y == z*z])`
- This produces $(3,4,5)$ and $(4,3,5)$ which are basically same triple
- Also this produces $(0,1,1)$ or $(4,0,4)$ etc. which looks meaningless for a right-angled triangle
- To overcome the first problem, we modify the code as follows:
`print([(x,y,z) for x in range(50) for y in range(x,50) for z in range(y,50) if x*x + y*y == z*z])`
- To overcome the second problem, we modify the code as follows:
`print([(x,y,z) for x in range(1,50) for y in range(x,50) for z in range(y,50) if x*x + y*y == z*z])`
- These generators which depend on earlier ones are called multiple generators

Example: Initializing matrix

- List comprehension notation is used in initializing matrix
- Example: Initialize a 5 X 4 matrix to all 0's
- 5 rows and 4 columns are stored row-wise
- `matrix = [[0 for i in range(4)] for j in range(5)]`
- `print(matrix)` would print as follows:
`[[0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0]]`
- `zero_column = [0 for i in range(4)]` # for column entry
- `zero_matrix = [zero_column for j in range(5)]` # zero_column 5 times
- `print(zero_matrix)` would also print the same as before
- However, if it is required to change the 0th row and 0th column to 9
- `zero_matrix[0][0] = 9` would update 0th column of all the rows to 9 because each row in `zero_matrix` points to the same list `zero_column`

Example: Computing an intersection

- Let a and b are two lists, in which no value is repeated (like a set)
- Construct a third list c containing their intersection (elements found in both the lists)
- Imperative approach # 1

```
c = []
for x in a:
    for y in b:
        if x == y:
            c.append(y)
print(c)
```

- Imperative approach # 2 (better than #1)

```
c = []
for x in a:
    if x in b:
        c.append(x)
print(c)
```

Example: Computing an intersection

- Functional programming approach # 1 (using lambda)

`c = filter(lambda x: x in b,a)`

- As the code is simpler, fewer chances to make mistakes
- It is relatively easy to look at the expression, understand what it is doing, and verify its correctness
- List comprehension method (even shorter than above)

`c = [x for x in a if x in b]`

- As a function

```
def c(a,b):  
    return [x for x in a if x in b]
```

Example: Find the Prime Numbers in a range

- Method known as ‘Sieve of Eratosthenes’ was first described by the Greek mathematician Eratosthenes in 300 BC
- (1) Start with a list of numbers from 2 to n, written in order
- (2) Select the first element as prime number
- (3) Then remove all values from the list which are a multiple of this number
- (4) Repeat from the step 2, until the list is empty
- A filter to eliminate all multiples of the first element can be written as

```
print(filter(lambda x: x % a[0] != 0, a[1:]))
```
- Alternatively, list comprehensions may be used

```
print [x for x in a[1:] if x % a[0] != 0]
```

Example: Find the Prime Numbers in a range

- Now the sieve function (recursive) may be defined as follows:

```
def sieve(lst):  
    if lst:      # check whether the list lst is empty  
        return lst[0:1] + list(sieve(list(filter(lambda x: x % lst[0] != 0, lst[1:]))))  
    else:  
        return []
```

`print(sieve(list(range(2, 25))))` will display

`[2, 3, 5, 7, 11, 13, 17, 19, 23]`

Example: Find the Prime Numbers in a range

- [] [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
- [2] [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
- [2, 3] [5, 7, 11, 13, 17, 19, 23, 25]
- [2, 3, 5] [7, 11, 13, 17, 19, 23]
- [2, 3, 5, 7] [11, 13, 17, 19, 23]
- [2, 3, 5, 7, 11] [13, 17, 19, 23]
- [2, 3, 5, 7, 11, 13] [17, 19, 23]
- [2, 3, 5, 7, 11, 13, 17] [19, 23]
- [2, 3, 5, 7, 11, 13, 17, 19] [23]
- [2, 3, 5, 7, 11, 13, 17, 19, 23] []

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 2: Effective Python
Lecture 16: 01/12/2021

Dr. Debranjan Sarkar

Module 2: Effective Python

Pythonic thinking and writing better Python code

- Programmers familiar with other languages normally try to write Python like C++, Java or whatever they know best
- New programmers may use the numerous concepts expressible in Python
- It is important for everyone to know the best way to do the most common things in Python
- The best way to write Python programs is to think in a Pythonic way
- The adjective ‘Pythonic’ is used to describe code that follows a particular style
- This style has emerged over time through experience using the language and working with others
- This style is not enforced by the compiler
- Python programmers prefer to
 - be explicit
 - choose simple over complex
 - maximize readability
- We shall discuss some tips to write better Python code

1. Know the version of
Python being used

Versions of Python

- Two major versions of Python are still in active use:
 - Python 2 and Python 3
- There are multiple popular runtimes for Pyhton
 - CPython, Jython, IronPython, PyPy etc.
- Be sure that the command-line for running Python on your system is your expected version
- Prefer Python 3 as this is currently the primary focus of the Python community

2. Follow the PEP 8 Style Guide

PEP 8 Style Guide

- Python Enhancement Proposal (PEP) # 8 is the style guide for formatting a Python code -- Always follow this guide
- Using a consistent style makes a code more approachable and easier to read by others (collaborative projects)
- Using a consistent style makes it easier to modify the code later by the developer
- Complete PEP-8 guide is available at <http://www.python.org/dev/peps/pep-0008> which gives coding conventions for the Python Code
- A few important rules of PEP-8 are given in the following

PEP 8 Style Guide: Code lay-out

- Use 4 spaces (instead of tabs) per indentation level
- Lines should be maximum 79 characters in length
- In a file, functions and classes should be separated by two blank lines
- In a class, methods should be separated by one blank line
- Do not put spaces around list indices, function calls or keyword argument assignments
- Put one (and only one) space before and after variable assignments
- Continuation of long expressions onto additional lines should be indented by four extra spaces from their normal indentation level

PEP 8 Style Guide: Naming

- Functions, variables, and attributes should be in lowercase_underscore format (e.g. `is_even()`, `student_age`)
- Protected instance attributes should be in `_leading_underscore` format
 - Protected members of a class can be accessed by other members within the class and are also available to their subclasses
 - No other entity can access these members
 - Suppose we have the following class which has protected attributes (`_alias`):
 - `class P:`
 - `def __init__(self, name, alias)`
 - `self.name = name` # Public
 - `self._alias = alias` # Protected
- Private instance attributes should be in `_double_leading_underscore` format
 - Private members of a class are only accessible within the class and not outside the class
 - Suppose we have the following class which has private attributes (`__alias`):
 - `class P:`
 - `def __init__(self, name, alias)`
 - `self.name = name` # Public
 - `self.__alias = alias` # Private
- Classes and exceptions should be in CapitalizeWord format (e.g. `class BankAccount`, `exception IndexError`)
- Module level constants should be in ALL_CAPS format (e.g. `PI = 3.14`, `GRAVITY = 9.8`)

PEP 8 Style Guide: Expressions and Statements

- Use inline negation (if a is not b) instead of negation of positive expressions (if not a is b)
- Do not check for empty values ([] or "") by checking the length (if len(lst) == 0). As empty values implicitly evaluate to False, use (if not lst)
- Similarly, for non-empty values like [2] or 'hello', (if lst) is implicitly True
- Avoid single-line 'if' statements, 'for' loops and 'while' loops, and 'except' compound statements. Spread these over multiple lines for clarity
- Always put 'import' statements at the top of the code
- Always use absolute names for modules when importing them, not names relative to the current module's own path
 - For example, to import xxx module from the yyy package, you should use (from yyy import xxx) and not just (import xxx)

3. Know the differences
between bytes, str, and unicode

bytes, str, and unicode

- To represent sequences of characters, there are two types in Python 3: bytes and str
- Instances of bytes contain raw 8-bit values, whereas, instances of str contain Unicode characters
- There are many ways to represent Unicode characters as binary data, the most common being UTF-8 (Unicode Transformation Format- 8-bit)
- UTF-8 is a variable-width character encoding, capable of encoding all 11,12,064 valid character code points in Unicode, using one to four one-byte code units
- UTF-8 is backward compatible with ASCII
- As there are two character types, two helper functions are required to convert between these two cases
- `to_str` method takes a str or bytes and always returns a str
- `to_bytes` method takes a str or bytes and always returns a bytes
- `print(to_str(b'abc'))` will display 'abc'
- `print(to_bytes(b'abc'))` will display b'abc'
- `print(to_str('abc'))` will display 'abc'
- `print(to_bytes('abc'))` will display b'abc'

Methods `to_str` and `to_bytes`

```
def to_str(bytes_or_str):  
    if isinstance(bytes_or_str, bytes):  
        value = bytes_or_str.decode('utf-8')  
    else:  
        value = bytes_or_str  
    return value      # Instance of str
```

```
def to_bytes(bytes_or_str):  
    if isinstance(bytes_or_str, bytes):  
        value = bytes_or_str  
    else:  
        value = bytes_or_str.encode('utf-8')  
    return value      # Instance of bytes
```

bytes, str, and unicode: Things to remember

- bytes and str instances are never equivalent – not even the empty string
- bytes and str instances cannot be used together with operators (> or +)
- Use helper functions to ensure that the inputs are of the expected type of character sequence (8-bit values, UTF-8 encoded characters, Unicode characters etc.)
- If it is required to read or write binary data to/ from a file, the file should always be opened using a binary mode (like ‘rb’ or ‘wb’)

```
f = open('test.bin', 'w') # will open the file in string write mode
```

```
f.write(b'xyz') # will give rise to the following exception
```

TypeError: write() argument must be str, not bytes

```
f = open('test.bin', 'wb') # will open the file in binary write mode
```

```
f.write(b'xyz') # will write binary data to the file
```

```
f.close() # will close the file
```

```
g = open('test.bin', 'rb') # will open the file in binary read mode
```

```
print(g.read()) # will read the binary data from the file and display b'xyz'
```

```
close(g) # will close the file
```

4. Know how to slice
sequences

Slicing of sequences

- The simplest uses for slicing are the built-in types list, str, and bytes
- The syntax is `somelist[start:end]`, where start is inclusive and end is exclusive
- Consider a list 'a' containing 8 elements
- When slicing from the start of a list, the 0 index should be left out
 - Use `a[:5]` instead of `a[0:5]`
- When slicing upto the end of a list, the final index should be left out
 - Use `a[5:]` instead of `a[5:len(a)]`
- Using negative numbers for slicing is helpful for doing offsets relative to the end of a list
 - From first to last-but-1 `a[:-1]`
 - Last three elements `a[-3:]`
 - From last-but-3 to last-but-1 `a[-3:-1]`
- If start and end indexes are beyond the boundaries of the list, a maximum length is established to consider for an input sequences. Example: `first_thirty = a[:30]`
- Accessing the same index directly causes an exception

`a[30]`

will give rise to

`IndexError: list index out of range`

Slicing of sequences

- Result of slicing a list is a whole new list
- References to the objects from the original list are maintained
- Modifying the result of slicing would not affect the original list

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
b = a[3:]; print(b)                      # will display ['d', 'e', 'f', 'g', 'h']
b[1] = 9; print(b)                        # will display ['d', 9, 'f', 'g', 'h']
print(a)                                  # will display ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

- When used in assignments, slices will replace the specified range in the original list
- The list will grow or shrink to accommodate the new values

```
print(a)                                # will display ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[3:7]= [9, 2]; print(a)                  # will display ['a', 'b', 'c', 9, 2, 'h']
```

Slicing of sequences

- If both the start and the end indexes are left out when slicing, a copy of the original list is obtained
- `b = a[:]; assert b == a and b is not a`
- If a slice is assigned with no start or end indexes, its entire contents will be replaced with a copy of what is referenced, instead of allocating a new list
- `a = ['a', 'b', 'c', 'd']; b = a; print(a)` will display `['a', 'b', 'c', 'd']`
- `a[:] = [11, 21, 31]; assert a is b; print(a)` will display `[11, 21, 31]`

5. Avoid using start, end, and stride in a single slice

start, end, and stride in slicing

- Python has a special syntax for the stride of a slice in the form somelist[start:end:stride]
- This facilitates to take every n^{th} item when slicing a sequence
- The stride makes it easy to group by even and odd indexes in a list

```
a = ['red', 'blue', 'yellow', 'green', 'white', 'black', 'orange', 'purple']
print(a[::-2]) will display      ['red', 'yellow', 'white', 'orange']
print(a[1::2]) will display      ['blue', 'green', 'black', 'purple']
```

- Slicing a byte string with a stride of -1 would reverse the byte string
`x = b'python'; print(x[::-1])` would display `b'nohtyp'`
- This works well for byte strings and ASCII characters but breaks for Unicode characters encoded as UTF-8 byte strings
- `w = 'আমরা'; x = w.encode('utf-8'); y = x[::-1]; z = y.decode('utf-8')` will give rise to the following error

`UnicodeDecodeError: 'utf-8' codec can't decode byte 0xbe in position 0: invalid start byte`

start, end, and stride in slicing

a = ['red', 'blue', 'yellow', 'green', 'white', 'black', 'orange', 'purple']

print(a[::-2]) will display ['red', 'yellow', 'white', 'orange']

print(a[::2]) will display ['purple', 'black', 'green', 'blue']

- ::2 selects every second item starting at the beginning
- ::-2 selects every second item starting at the end and moving backwards

print(a[2::2]) will display ['yellow', 'white', 'orange']

print(a[-2::-2]) will display ['orange', 'white', 'yellow', 'red']

print(a[-2:2:-2]) will display ['orange', 'white']

print(a[2:2:-2]) will display []

Avoid using start, end, and stride in a single slice

- Specifying start, end and stride in a slice can be highly confusing
- Prefer using positive stride values in slices without start or end indexes
- Avoid negative stride values, if possible
- Avoid using start, end and stride together in a single slice
- If all three parameters are required, use two assignments:
 - one for slice
 - another for stride as shown below:
- `b = a[::-2] # ['red', 'yellow', 'white', 'orange']`
- `c = b[1:-1] # ['yellow', 'white']`

6. Use List Comprehensions
instead of map and filter

List Comprehensions vs map

- To derive one list from another, we use the expressions, known as list comprehensions
- Example to derive a list, each element of which is the square of the corresponding element of the parent list
- `a = [1, 2, 3, 4, 5, 6, 7]`
- `squares = [x**2 for x in a] # will produce [1, 4, 9, 16, 25, 36, 49]`
- Built-in function ‘map’ can also do the same thing, using a lambda function:
`squares = map(lambda x: x ** 2, a)`
- List comprehensions are clearer than map built-in function
- ‘map’ requires creating a lambda function for the computation, which is visually noisy

List Comprehensions vs filter

- Unlike map, list comprehensions can filter items from the input list, removing corresponding outputs from the result
- Example to derive a list which contains the squares of only those numbers which are divisible by 2 in the parent list
- `even_squares = [x**2 for x in a if x % 2 == 0]`
- `print(even_squares)` will display [4, 16, 36]
- The built-in function ‘filter’ can be used alongwith map to achieve the same outcome, but it is much harder to read
- `alt = map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, a))`
- `assert even_squares == list(alt)`
- Dictionaries and sets have their own equivalents of list comprehensions
- ======Points to note=====
- List comprehensions are clearer than map and filter as they do not require extra lambda function
- List comprehensions allow to easily skip items from the input list, which map cannot do without the help of filter
- Dictionaries and sets also support list comprehension schemes

7. Avoid more than two
expressions
in List Comprehensions

List Comprehensions: Beyond basic usage

- List comprehensions support multiple levels of looping
- Example #1: To simplify a matrix into one flat list of all cells
- Two ‘for’ expressions run in the order provided from left to right
- `matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
- `flat = [x for row in matrix for x in row]`
- `print(flat)` will display [1, 2, 3, 4, 5, 6, 7, 8, 9]
- Example #2: To square the value in each cell of a 2-D matrix
- `squared = [[x**2 for x in row] for row in matrix]`
- `print(squared)` will display [[1, 4, 9], [16, 25, 36], [49, 64, 81]]
- This expression is noisier because of extra [] characters, but it is still easy to read
- If this expression included another loop, the list comprehension would become much more complicated
- Same result can be produced using normal loop statements, which is clearer than list comprehension

List Comprehensions: Beyond basic usage

- List comprehensions also support multiple if conditions
- Example #3: To filter a list of numbers to only even values greater than 4
- The following two list comprehensions are equivalent
- `a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
- `b = [x for x in a if x > 4 if x % 2 == 0]`
- `c = [x for x in a if x > 4 and x % 2 == 0]`
- Example #4: To filter a matrix so that the only cells remaining are those divisible by 3 in rows that sum to 10 or higher
- `matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
- `filtered = [[x for x in row if x % 3 == 0] for row in matrix if sum(row) >= 10]`
- `print(filtered)` will display `[[6], [9]]`
- This list comprehension is short but extremely difficult to read

List Comprehensions: Things to remember

- List comprehensions support multiple levels of loops and multiple conditions per loop level
- List comprehensions with multiple expressions are very difficult to read
- The rule of thumb is to avoid using more than two expressions in a list comprehension
- This could be two conditions, two loops, or one condition and one loop
- If it gets more complicated than this, normal if and for statements should be used and a helper function should be written

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 2: Effective Python
Lecture 17: 06/12/2021

Dr. Debranjan Sarkar

8. Consider Generator
expressions for Large
comprehensions

Large comprehensions

- List comprehensions may create whole new list containing one item for each value in the input sequence
- For large inputs, this could consume significant amounts of memory and might cause the program to crash
- Example: To read a file and return the number of characters in each line
- Doing this with a list comprehension would require holding the length of every line of the file in memory
- If the file is enormous, list comprehensions are problematic
- The following list comprehension can only handle small input values
- `value = [len(x) for x in open('C:/Users/Admin/Desktop/test.txt')]`
- `print(value)` will display [22, 40, 38, 96, 99, 37]
- For very large files, we use Generator Expressions

Generator expressions

- Generator expressions are the generalization of list comprehensions and generators
- Generator expressions do not materialize the whole output sequence when they are run
- Rather, generator expressions evaluate to an iterator that yields one item at a time from the expression
- A list-comprehension-like syntax is put between () characters to create a generator expression
- The following is the generator expression for the above problem
- `it = (len(x) for x in open('C:/Users/Admin/Desktop/test.txt'))`
- This evaluates to an iterator and does not make any forward progress
- `print(it)` will display `<generator object <genexpr> at 0x0334BA00>`

Generator expressions

- The returned iterator can be advanced (by using the built-in function ‘next’) one step at a time to produce the next output from the generator expression as needed
- `print(next(it))` will display 22
- `print(next(it))` will display 40
- Now the code will consume as much of the generator expression as you want without risking a blow up in memory usage
- Generator expressions can be composed together
- In the following, the iterator (it), returned by the above generator expression, is used as the input of another generator expression
- `roots = ((x, x**0.5) for x in it)`
- Each time the iterator is advanced, it will advance the interior iterator also
- `print(next(roots))` will display (38, 6.164414)
- Chaining generators like this execute very quickly in Python

Generator expressions: Important points

- List comprehensions can cause problems for large inputs by using too much memory
- Generator expressions avoid memory issues by producing outputs one at a time as an iterator
- Generator expressions can be composed by passing the iterator from generator expression into the ‘for’ subexpression of another
- Generator expressions execute very quickly when chained together

9. Prefer enumerate over range

‘enumerate’ versus ‘range’

- Let us relook into the use of function range

```
color_lst = ['Red', 'Blue', 'Yellow']           >>>
for i in range(len(color_lst)):
    color = color_lst[i]
    print('%d. %s' %(i +1, color))               1. Red
                                                2. Blue
                                                3. Yellow
```

- Python provides a function enumerate which does the same in a much clearer way
- c = enumerate(color_lst)
- print(c) will display <enumerate object at 0x01E27808>
- print(next(c)) will display (0, 'Red')
- The above code may be written using enumerate as follows:

```
color_lst = ['Red', 'Blue', 'Yellow']
for i, color in enumerate(color_lst):
    print('%d. %s' %(i +1, color))
```

Points to remember on enumerate over range

- Enumerate provides a concise syntax for looping over an iterator (an object that contains a countable number of values) and getting the index of each item from the iterator
- Prefer enumerate instead of looping over a range and indexing into a sequence
- A second parameter may be supplied to the function enumerate to specify the number from which to begin counting (zero is the default)

10. Use zip to process
iterators in Parallel

Recapitulation of list comprehension

- List comprehensions make it easy to take a source list and get a derived list by applying an expression

```
cities = ['Kolkata', 'Hyderabad', 'Delhi']
```

```
letters = [len(c) for c in cities]
```

- To find out the city with longest name, we have to iterate over both lists in parallel
- To do that, we can iterate over the length of the cities source list, as shown below:

```
longest_name = None  
max_letters = 0  
for i in range(len(cities)):  
    count = letters[i]  
    if count > max_letters:  
        longest_name = cities[i]  
        max_letters = count  
print(longest_name)
```

```
>>>
```

```
Hyderabad
```

Using enumerate

- The problem is that the indexes into cities and letters make the code hard to read
- Indexing into the arrays (cities and letters) by the loop index *i* happens twice
- Using enumerate improves this slightly

```
cities = ['Kolkata', 'Hyderabad', 'Delhi']
letters = [len(c) for c in cities]
longest_name = None
max_letters = 0
for i, city in enumerate(cities):
    count = letters[i]
    if count > max_letters:
        longest_name = city
        max_letters = count
print(longest_name)
```

Using zip

- cities = ['Kolkata', 'Hyderabad', 'Delhi']
- letters = [len(c) for c in cities]
- z = zip(cities, letters); print(z) will display `<zip object at 0x02D07828>`
- print(next(z)) will display ('Kolkata', 7)
- To make this code more clear, one can use zip built-in function

```
cities = ['Kolkata', 'Hyderabad', 'Delhi']
letters = [len(c) for c in cities]
longest_name = None
max_letters = 0
for city, count in zip(cities, letters):
    if count > max_letters:
        longest_name = city
        max_letters = count
print(longest_name)
```

- The problem of zip function is that it behaves strangely if the input iterators are of different lengths
- However, zip is the preferred function to be used to process the iterators in parallel

11. Avoid else blocks after
for and while loops

‘else’ after a ‘for’ or a ‘while’ loop

- ‘else’ statement after the for or while loop indicates that the loop (for or while) was terminated naturally

- def find_position(lst, val):

```
# pos = -1           this statement has been commented
for i in range(len(lst)):
    if lst[i] == val:
        pos = i
        break
else:
    # This statement is executed when the for loop was terminated naturally (i.e. val was
    # not found in lst
    pos = -1
return(pos)
```

Avoid else blocks after for and while loops

- The else part of the for/else is executed only when the for loop was completed
- Using a break statement in a loop will actually skip the else block

```
for i in range(3):
```

```
>>>
```

```
    print('Loop %d' % i)
```

```
Loop 0
```

```
    if i == 1:
```

```
Loop 1
```

```
        break
```

```
else:
```

```
    print('Else block')
```

- The else block will run immediately if it is looped over an empty sequence

```
for i in []:
```

```
>>>
```

```
    print('Looping over an empty sequence')
```

```
Else block after for loop
```

```
else:
```

```
    print('Else block after for loop')
```

Avoid else blocks after for and while loops

- The else block also runs when while loops are initially false

while False:

```
    print('This statement is never executed')
```

else:

```
    print('Else block after while loop')
```

>>>

```
Else block after while loop
```

- else block after loops is useful when the loops are used to search for something
- The else block after a loop runs only if the loop body did not encounter a break statement and the loop was terminated naturally
- Avoid using else blocks after loops because their behaviour is not intuitive and can be confusing

12. Take advantage of each block in try/ except/ else/ finally

Exception handling

- try:
 -
 -
 - except IndexError:
 -
 -
 - except (NameError, KeyError):
 -
 -
 - except:
 -
 -
 - else:
 -
 -
- # if error is predicted in this block
- # when index error occurs
- # when there is a name error or key error
- # otherwise (for all other exceptions)
- # if no errors were raised
(try block is executed successfully)
(similar to else in for or while)

Take advantage of each block in try/ except/ else/ finally

- **try/ except/ else/ finally block**
- **Try:** First try block (the code between try and except clause) is executed
 - This block will test whether there is any exception generated while execution
 - If there is no exception, then only try clause will run, except clause is finished.
- **Except:** If any exception occurs, the try clause will be skipped and except block will run
 - A try statement can have more than one except clause
 - Here one can handle the error
- **Else:** If there is no exception, this block will be executed
- **Finally:** This block will always be executed whether any exception is generated or not

Example of try/ except/ else/ finally block

```
# Python code to show working of try-except-else-finally
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        print("Sorry, you are dividing by zero !!")
    else:
        print("Your answer is : ", result)
    finally:
        print("This is always printed")
```

Floor division – gives only integer part as the answer
if no errors were raised (try block is executed successfully)
This block is always executed, irrespective of the exception

- `divide(7, 3)` will display
- `divide(7, 0)` will display

Your answer is : 2
This is always printed
Sorry, you are dividing by zero !!
This is always printed

13. Prefer Exceptions to Returning None

Returning None from a function

- Returning None from a function makes sense in some cases
- A function may divide two numbers and returns the result (or None if the result is undefined)

```
def divide(a,b):  
    try:  
        return a / b  
    except ZeroDivisionError:  
        return None
```

```
result = divide(x,y)  
if not result:  
    print('Invalid Inputs')
```

- If the numerator is zero and the denominator is not 0, then the result is 0 and the first code above is wrong !!
- This is a common mistake in Python, when None has a special meaning
- This is why returning None from a function is error prone

```
result = divide(x,y)  
if result is None:  
    print('Invalid Inputs')
```

Returning None from a function

- There are two ways to reduce the chance of such errors
- (1) Split the return value into two-tuple
 - flag to indicate success or failure
 - Result or None if there is an error

```
def divide(a,b):  
    try:  
        return True, a / b  
    except ZeroDivisionError:  
        return False, None
```

```
success, result = divide(x,y)  
if not success:  
    print('Invalid Inputs')
```

- The problem is that the callers can easily ignore the first part of the tuple and it is as bad as just returning None

Returning None from a function

- (2) Better way to reduce the errors is not to return None at all
- Instead, an exception is raised upto the caller who deals with the exception

```
def divide(a,b):  
    try:  
        return a / b  
    except ZeroDivisionError as e:  
        raise ValueError('Invalid inputs') from e
```

- Now caller should handle the exception for the invalid input case

```
x, y = 5, 2  
try:  
    result = divide(x, y)  
except ValueError:  
    print('Invalid inputs')  
else:  
    print('Result is ', result)
```

```
>>>
```

```
Result is 2.5
```

Points to remember

- Functions that return None to indicate special meaning are error prone because None and other values (e.g. zero, empty string) all evaluate to False in conditional expressions
- Raise exceptions to indicate special situations instead of returning None
- Expect the calling code to handle the exceptions properly when these are documented

Reference

- Effective Python, Brett Slatkin, Pearson

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 1: Data Science Introduction
Lecture 18: 07/12/2021

Dr. Debranjan Sarkar

Module 1: Data Science Introduction

Defining Big Data and Data Science

- *Big data* is a collection of very large and complex data sets
- It is difficult to process Big Data using traditional data management techniques (viz. RDBMS)
- *Data science* deals with methods for analysis of massive amount of data and extraction of the knowledge from the data
- Both Data science and big data evolved from statistics and traditional data management but are now considered to be distinct disciplines

Defining Big Data and Data Science

- The characteristics of big data are often referred to as the four Vs:
- *Volume*—How much data is there?
- *Variety*—How diverse are different types of data?
- *Velocity*—At what speed is new data generated?
- *Veracity*—How accurate is the data? (Veracity means Truth or Accuracy)
- Big data is different from the data found in traditional data management tools (RDBMS) because of these properties
- So, data capture, curation, storage, search, sharing, transfer, and visualization – all the aspects of big data are really challenging
- In addition, big data requires specialized techniques to extract the insights
- In Data science and Big data, many different types of data are encountered
- Each type of data requires different tools and techniques

Facets of data

- Following are the main categories of data types:
 - 1. Structured
 - Names, Addresses, Aadhaar numbers, Credit card numbers, etc. (Excel sheet, SQL database)
 - 2. Unstructured
 - A regular email, audio, video, social media postings etc.
 - 3. Natural language
 - A human-written email is a perfect example of natural language data
 - 4. Machine-generated
 - Web server logs, Call detail records, Network event logs, Telemetry etc.
 - 5. Graph-based
 - Friends in a social media network
 - 6. Audio, video, and images
 - 7. Streaming
 - “What’s trending” on Twitter, Live sports or music events, Stock market etc.

1. Structured Data

- Data is said to be structured when it adheres to a pre-defined data model and is organized in a pre-defined manner
- Structured data resides in a fixed field within a record and can be analyzed in a straightforward manner
- Structured data is highly organized and easily understood by machine language
- Examples of structured data include names, dates, addresses, aadhaar numbers, credit card numbers, geolocation, etc. etc.
- Structured data conforms to a tabular format with relationship between the different rows and columns
- Common examples of structured data are Excel tables or SQL databases.
- SQL, or Structured Query Language, is the preferred way to manage and query data that resides in databases
- Hierarchical data such as a family tree is an example of structured data which is difficult to store in a traditional relational database
- In general, whatever data comes spontaneously, is unstructured but humans and machines make them structured

2. Unstructured Data

- Data that either does not have a predefined data model or is not organized in a pre-defined manner is called Unstructured data
- As the content of unstructured data is context-specific or varying, it cannot be (or is difficult to be) fit into a pre-defined data model
- Unstructured data is usually not easily searchable
- Examples include formats like a regular email, audio, video, and social media postings
- Although email contains structured elements such as the sender, title, and body text, it is extremely difficult to find the number of people who have written an email complaint about a specific employee
 - This is because there are many ways to refer to a person in a mail
 - Moreover, so many languages and dialects complicate the issue

3. Natural language Data

- Natural language is a special type of unstructured data
- It requires knowledge of both the specific data science techniques and linguistics to process natural language
- Though there has been a lot of developments in the Natural Language understanding and natural language generation, even state-of-the-art techniques are unable to decipher the meaning of every piece of text
- As natural language is ambiguous by nature, it is sometimes difficult even for human beings to understand natural language unambiguously
- The concept of meaning itself is questionable here
- When two people listen to the same conversation, they may not get the same meaning
- The meaning of the same words can vary when coming from someone upset or joyous
- A human-written email is a perfect example of natural language data

4. Machine-generated Data

- Machine-generated data are automatically created by a computer, process, application, or other machine without human intervention
- Machine-generated data is becoming a major data resource and will continue to do so
- *The internet of things* has resulted in several times more connected ‘things’ than ‘people’
- The analysis of machine data relies on highly scalable tools, due to its high volume and speed
- Examples of machine data are web server logs, call detail records, network event logs, and telemetry
- The machine data would fit nicely in a classic table-structured database

5. Graph-based or Network Data

- A graph is a mathematical structure to model pair-wise relationships between objects
- Graph data or network data is the data that focuses on the relationship or adjacency of objects
- The graph structures use nodes, edges, and properties to represent and store graphical data
- Friends in a social media network are an example of graph-based data
- Graph structure allows us to calculate specific metrics such as the influence of a person and the shortest path between two people
- The power and sophistication comes from multiple, overlapping graphs of the same nodes
- Assume that the connecting edges of a graph of a few people show “friends” on Facebook
- Another graph with the same people which connects business colleagues via LinkedIn
- And a third graph based on movie interests on Netflix
- Overlapping the three different-looking graphs makes more interesting questions possible
- Graph databases are used to store graph-based data and are queried with specialized query languages such as SPARQL

6. Audio, video, and images

- Audio, video, and image are data types that pose specific challenges to a data scientist
- Tasks that are trivial for humans, such as recognizing objects in pictures, or voice in an audio, turn out to be challenging for computers
- High-speed cameras at stadiums may capture movements of ball and players for the purpose of live, in-game analytics
- Algorithms have been developed which are capable of learning how to play video games
- These algorithms take the video screen as input and learn to interpret everything via a complex process of deep learning
- The learning algorithm takes in data as it is produced by the computer game: it is streaming data

7. Streaming data

- Streaming data can take any of the previous forms
- But this data flows into the system when an event happens instead of being loaded into a data store in a batch
- It is not really a different type of data
- We need to adapt our process to deal with this type of streaming information
- Examples of streaming data:
 - “What’s trending” on Twitter
 - Live sporting or music events, and
 - Stock market

The Data Science Process: An overview

The Data Science Process: an overview

- *Data science* deals with methods for analysis of massive amount of data (big data) and extraction of the knowledge from the data
- Following a structured approach to data science helps maximize the chances of success in a data science project at the lowest cost
- It also makes it possible to take up a project as a team, with each team member focusing on what they do best
- However, this approach may not be suitable for every type of project or be the only way to do good data science
- The data science process typically consists of six steps
 1. Setting the research goal
 2. Retrieving data
 3. Data preparation
 4. Data exploration
 5. Data modelling
 6. Presentation and automation

1. Data Science Process: Setting the research goal

- Data science is mostly applied in the context of an organization
- When a data science project is taken up, the first step of this process is setting a *research goal and preparing a project charter*
- The main purpose here is to make sure that all the stakeholders understand the *what, how, and why* of the project
- The outcome should be a clear research goal, a good understanding of the context, well-defined deliverables, and a plan of action with a timetable
- This information is then best placed in a project charter
- This charter contains information such as what is the topic of research, how the company benefits from that, what data and resources are needed, a timetable, and deliverables
- The charter is used to make an estimation of the project costs and the data and people required for the project to become a success

2. Data Science Process: Retrieving data

- The second step is data retrieval, or to collect data for analysis
- Data may be in-house or may be delivered by a third-party and takes many forms ranging from Excel spreadsheets to different types of databases
- In the project charter, it has been stated which data is needed and where it is available
- In this step it is ensured that the data from its owner can be used in the program
- In other words, it is checked that the data exists, the quality of data is good and it can be accessed for use
- The result is data in its raw form, which probably needs polishing and transformation before it becomes usable
- Points to remember in this step
 - Start with data stored within the company
 - Don't be afraid to shop around
 - Do data quality checks now to prevent problems later

3. Data Science Process: Data preparation

- Data collection is an error-prone process
- The raw data is to be prepared to enhance its quality and prepare it for use in subsequent steps
- This phase consists of three subphases:
 - *data cleansing* detects and corrects different kinds of errors in the data and removes false values from the data source and inconsistencies across data sources
 - *data integration* enriches data sources by combining information from multiple data sources, and
 - *data transformation* ensures that the data is in a suitable format which is directly usable in your models
- After successful completion of Data Preparation step, one can progress to data visualization and modeling

4. Data Science Process: Data exploration

- The goal of Data exploration step is to gain a deep understanding about the data
- This step is also known as EDA (Exploratory Data Analysis)
- In this step, we try to understand how variables interact with each other, the distribution of the data, and whether there are outliers (not fit in the dataset)
- To achieve this, we mainly use Descriptive Statistics, Visual Techniques and Simple Modeling
- Descriptive statistics
 - mean, median, mode, standard deviation etc.
 - Example Grade Point Average (GPA) of a student

4. Data Science Process: Data exploration

- Visual techniques
- Information becomes much easier to grasp when shown in a picture
- So we use graphical techniques to gain an understanding of the data and the interactions between variables
- Some visualization techniques used in this phase:
 - Simple line graphs
 - Composite graph from simple graphs
 - Animated graph or interactive graph
 - Histograms (Example: Number of people in the age groups of 5-year intervals)
- Simple modeling
 - Tabulation, clustering, and other modeling techniques can also be a part of data exploration
 - Building simple models can be a part of this step
- The insights we gain (a good grasp of data) from this phase will enable us to build models

5. Data Science Process: Data modeling or model building

- In this phase we use models, domain knowledge, and insights about the data that was found in the previous steps to answer the research question
- The goal of this step is to make better predictions, classify objects, or gain an understanding of the system that are being modeled
- We select a technique from the fields of statistics, machine learning, data mining, operations research, and so on
- The components of model building (which is an iterative process) are as follows:
 1. Selection of a modeling technique and variables to enter in the model
 2. Execution of the model
 3. Diagnosis and model comparison

6. Data Science Process: Presentation and automation

- The last step of the data science model is to present the results to the stakeholders and automate the analysis, if needed (for repetitive reuse and integration with other tools)
- The results can take many forms, ranging from presentations to research reports
- For certain projects, it is required to perform the process over and over again, so automation will save time
- One goal of the project is to change a process and/or make better decisions
- It is required to convince the stakeholders that the findings will indeed change the business process as expected
- In this stage, the *soft skills* will be most useful and extremely important

The Big Data Ecosystem

The Big Data Ecosystem

- The big data ecosystem can be grouped into technologies that have similar goals and functionalities
- The components of the big data ecosystem are as follows:
 - 1. Distributed file systems
 - 2. Distributed programming framework
 - 3. Data integration framework
 - 4. Machine learning frameworks
 - 5. NoSQL databases
 - 6. Scheduling tools
 - 7. Benchmarking tools
 - 8. System deployment
 - 9. Service programming
 - 10. Security

1. Distributed file systems

- A *distributed file system* is similar to a normal file system, except that it runs on multiple servers simultaneously
- As it is a file system, one can do almost everything as can be done on a normal file system
- Every file system, including the distributed files system, can take actions such as storing, reading, and deleting files and adding security to files
- Advantages of Distributed file systems:
 - They can store files larger than any one computer disk
 - Files get automatically replicated across multiple servers for redundancy or parallel operations while hiding the complexity of doing so from the user
 - The system scales easily: you're no longer bound by the memory or storage restrictions of a single server

1. Distributed file systems

- In the past, scale was increased by augmenting the server with more memory, storage, and a better CPU (vertical scaling)
- Nowadays another small server can be added (horizontal scaling)
- This principle makes the scaling potential virtually limitless
- Presently, the best-known and most common distributed file system is the *Hadoop File System (HDFS)*, which is an open-source implementation of the Google File System
- However, many other distributed file systems exist:
 - Red Hat Cluster File System
 - Ceph File System
 - Tachyon File System, etc.

2. Distributed programming framework

- To exploit the advantages of stored data on a distributed hard disk in a distributed file system
 - we would not move our data to the program
 - rather we would move our program to the data
- When we start from scratch with a normal general-purpose programming language (e.g. C, Python, or Java), we have to deal with the complexities that come with distributed programming
- These include
 - restarting the failed jobs
 - tracking the results from the different subprocesses, and so on
- However, the open-source community has developed many frameworks to handle this, and these give a much better experience working with distributed data and dealing with many of the challenges it carries

3. Data integration framework

- In a distributed file system environment,
 - we need to add data
 - we need to move data from one source to another
- The data integration frameworks are required for this purpose
- Example of Data Integration Network:
 - Apache Sqoop
 - Apache Flume excel
- The process is similar to an extract, transform, and load process in a traditional data warehouse

4. Machine learning frameworks

- We gain a good grasp of data in the data exploration phase which enables us to build models
- Here, we rely on the fields of machine learning, statistics, and applied mathematics
- To do some scientific computation in earlier days, it was only needed to derive the mathematical formulas, write them in an algorithm, and load the data
- With the enormous amount of data available nowadays, one computer can no longer handle the workload by itself
- In fact, several algorithms developed in the previous millennium would never terminate before the end of the universe, even if every computer available on Earth is used
- An example is trying to break a password by testing every possible combination

4. Machine learning frameworks

- One of the biggest issues with the old algorithms is that they do not scale well
- With the amount of data we need to analyze today, this becomes problematic, and specialized frameworks and libraries are required to deal with this amount of data
- The most popular machine-learning library for Python is Scikit-learn
- There are many other Python libraries:
 - *PyBrain for neural networks*—Neural networks are learning algorithms that mimic the human brain
 - *NLTK or Natural Language Toolkit*—Its focus is working with natural language
 - *Pylearn2*—Another machine learning toolbox but a bit less mature than Scikit-learn
 - *TensorFlow*—A Python library for deep learning provided by Google

5. NoSQL databases

- When we want to store huge amounts of data, a software is required that is specialized in managing and querying this data
- Traditionally, relational databases such as Oracle SQL, MySQL, Sybase IQ, and others were used
- Traditional databases had shortcomings that did not allow them to scale well
- Though these are still used, new types of databases, called NoSQL (Not only SQL), have emerged
- By solving several of the problems of traditional databases, NoSQL databases allow for a virtually endless growth of data
- These shortcomings relate to every property of big data: their storage or processing power can't scale beyond a single node and they have no way to handle streaming, graph, or unstructured forms of data

5. NoSQL databases

- Many different types of databases have arisen
- They can be categorized into the following types:

Column databases

Document stores

Streaming data

Key-value stores

SQL on Hadoop

Graph databases

6. Scheduling tools

- Scheduling tools help to automate repetitive tasks and to trigger jobs based on events such as adding a new file to a folder
- These are similar to tools such as CRON on Linux but are specifically developed for big data
 - CRON is used to schedule jobs to run periodically at fixed times, dates, or intervals
 - It typically automates system maintenance or administration and may be useful for downloading files from internet or downloading emails at regular intervals
- One can use these scheduling tools, for instance, to start a MapReduce task whenever a new dataset is available in a directory
 - Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Reduce task takes the output from a map and combines those data tuples into a smaller set of tuples

7. Benchmarking tools

- This class of tools was developed to optimize a big data installation by providing standardized profiling suites
- A profiling suite is taken from a representative set of big data jobs
- Benchmarking and optimizing the big data infrastructure and configuration are generally not the jobs for data scientists but for a professional, specialized in setting up IT infrastructure
- Using an optimized infrastructure can make a big cost difference

8. System deployment

- Setting up of a big data infrastructure is not an easy task
- System deployment tools assist engineers in deploying new applications into the big data cluster
- They mainly automate the installation and configuration of big data components
- This is not a core task of a data scientist

9. Service programming

- Suppose that you have made a world-class soccer prediction application
- You would like to allow others to use the predictions made by your application
- However, you have no idea of the architecture or technology of everyone keen on using your predictions
- Service tools excel here by exposing big data applications to other applications as a service
- Data scientists sometimes need to expose their models through services
- Example: REST service (REST stands for representational state transfer)
- It is often used to feed websites with data

10. Security

- In order to limit the access of everybody to all of the data, security tools are required
- Sometimes, fine-grained control (each data item has its own access control policy)
- Big data security tools allow us to have central and fine-grained control over access to the data
- Big data security has become a topic in its own right
- Data scientists are usually only confronted with it as data consumers but normally they do not implement the security themselves

Reference

- Introducing Data Science, by Davy Cielen, Arno D. B. Meysman, and Mohamed Ali: Manning Publications Co.

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Processing with NumPy
Lecture 19: 08/12/2021

Dr. Debranjan Sarkar

The basics of NumPy Arrays

- NumPy is the short form of “Numeric Python”
- It is a package that contains several classes, functions, variables etc. to deal with scientific calculations in Python
- NumPy is useful to create and process single and multi-dimensional arrays
- NumPy array is a central data structure of the NumPy library
- One of the tools in NumPy is a high-performance multidimensional array object that is a powerful data structure for efficient computation of arrays and matrices
- NumPy also contains a large library of mathematical functions viz.
 - Linear algebra functions
 - Fourier Transforms
- NumPy contains both an array class and a matrix class

Array versus matrix in NumPy

- The array class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing
- The matrix (2-dimensional array) is intended to facilitate linear algebra computations specifically
- For array, * [or, multiply()] means element-wise multiplication and @ [or, dot()] means matrix multiplication
- For matrix, * means matrix multiplication and for element-wise multiplication, one is required to use the multiply() function
- For array, the vector shapes $1 \times N$, $N \times 1$ and N are all different
- For matrix, one dimensional arrays are always upconverted to $1 \times N$ or $N \times 1$ matrices

The basics of NumPy Arrays

- To work with NumPy, we have to import ‘numpy’ module into Python program
- import numpy as np
- arr0 = np.array([1, 2, 3, 4]) # 1D array, shape = (4,)
- arr1 = np.array([[1, 2, 3, 4]]) # 2D array, shape = (1,4)
- arr2 = np.array([[1], [2], [3], [4]]) # 2D array, shape = (4,1)
- arr3 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]) # 2D array, shape = (2,4)
- arr4 = np.array([[[1, 2, 3, 4], [5, 6, 7, 8]], [[4, 3, 2, 1], [8, 7, 6, 5]], [[0, 1, 0, 1], [1, 0, 1, 0]]]) # 3D array, shape = (3,2,4)
- print(arr0.shape) prints the shape of arr0
- Other methods of creating one dimensional arrays:
 - arr = np.array([1.5, 2, 4.7], float) # 1D array of ‘float’ numbers
 - arr = np.linspace(1,2,3) # generates 3 equi-spaced numbers between 1 and 2
in linear scale ([1.0, 1.5, 2.0])
 - arr = np.logspace(1,2,3) # generates 3 equi-spaced numbers between 1 and 2
in log scale ([10.0, 31.62, 100.0]) => log(31.62) = 1.5

Slicing and Indexing (1D arrays)

- Indexing refers to the locations of the elements
- Example: `arr = np.array([1, 7, 5, -4, 8, -3])`
- `arr[0] = 1` `arr[3] = -4` `arr[-1] = -3`
- Slicing refers to extraction of a range of elements from the array
- Format for slicing `arrayname[start : stop : stepsize]`
- `arr[1:5]` \Rightarrow `([7, 5, -4, 8])`
- `arr[1:6:2]` \Rightarrow `([7, -4, -3])`
- `arr[::-1]` \Rightarrow `([1, 7, 5, -4, 8, -3])`
- `arr[:]` \Rightarrow `([1, 7, 5, -4, 8, -3])`
- `arr[3:6]` \Rightarrow `([-4, 8, -3])`
- `arr[-1:-4:-1]` \Rightarrow `([-3, 8, -4])`
- # length = 6; start = -1 + 6 = 5; stop = -4 + 6 = 2; stepsize = - 1

Slicing and Indexing (1D arrays)

- Assigning scalar value to a slice as in `arr[2:5] = 2` will propagate the value to the entire selection and the array (`arr`) will now become
`([1, 7, 2, 2, 2, -3])`
- Array slices are **views** on the original array
- That is, the data is not copied, and any modifications to the view will be reflected in the source array
- `arr_slice = arr[2:5]` # `arr_slice` becomes `([2, 2, 2])`
- `arr_slice[1] = 100`
- `print(arr)` # will display `([1, 7, 2, 100, 2, -3])`
- `arr_slice[:] = 50`
- `print(arr)` # will display `([1, 7, 50, 50, 50, -3])`
- To have a copy of a slice of an ndarray instead of a view, we use
`arr[1:3].copy()`

Slicing and Indexing (2D arrays)

- Indexing: `arr[i][j]` or `arr[i, j]` => the element in the i^{th} row and j^{th} column of the array
- Example: `arr = np.array([[1,2,3],[4,5,6],[7,8,9]])`
- `arr[0][1] = 2` `arr[0,1] = 2` `arr[2][2] = 9` `arr[1][-1] = 6` `arr[2] = ([7,8,9])`

- Slicing: Format for slicing `arrayname[start:stop:stepsize]`
- To generate sequence from 11 to 30 and show the elements as 5 rows X 4 columns, we use the statement `arr = np.reshape(range(11,31), (5,4))`
- `print(arr)` will show the array as follows: `([[11,12,13,14], [15,16,17,18], [19,20,21,22], [23,24,25,26], [27,28,29,30]])`
- `print(arr[0:2, 0:3])` will display as follows: `([[11,12,13], [15,16,17]])`

Slicing and Indexing (2D arrays)

- In a 2-D array, axis 0 goes vertically down (indicating row number) and axis 1 goes horizontally right (indicating column number)
- `print(arr[:2, 1:])` will display as $\begin{bmatrix} [12, 13, 14] \\ [16, 17, 18] \end{bmatrix}$
- By slicing like this, we always obtain array views of the same number of dimensions
- By mixing integer indices and slices, we get lower dimensional slices
- `print(arr[1, :2])` will display as $([15, 16])$
- `print(arr[2, :1])` will display as $([19])$

2-D array slicing

- Let us consider a 3 X 3 array (arr):

1	2	3
4	5	6
7	8	9

- Expression
- arr[:2, 1:]

<u>Result</u>	<u>Shape</u>
2 3	(2, 2)
5 6	

-
- arr[2]
- arr[2, :]
- arr[2:, :]
-
- arr[:, :2]

7 8 9	(3,)
	(3,)
	(1, 3)

-
- arr[1, :2]
- arr[1:2, :2]
-

1 2	(3, 2)
4 5	
7 8	

4 5	(2,)
	(1, 2)

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Processing with NumPy
Lecture 20: 13/12/2021

Dr. Debranjan Sarkar

Generating arrays using built-in functions

- We have already seen `np.array()`, `np.linspace()`, `np.logspace()`
- `print(np.arange(1,10,2))` will display [1 3 5 7 9]
- `print(np.ones((2,4)))` will display [[1. 1. 1. 1.],
[1. 1. 1. 1.]]
- `print(np.zeros((2,4)))` will display [[0. 0. 0. 0.],
[0. 0. 0. 0.]]
- `print(np.random.rand(6))` will display the following
[0.31836133 0.28359678 0.79266958 0.48596108 0.5974501 0.54823155]
- `print(np.random.rand(6).reshape(2,3))` will display the following
[[0.78606042 0.19335324 0.2774488]
[0.59675646 0.79352972 0.22551927]]
- `a = np.zeros((2,4))`
- `print(a.shape)` will display the dimension of the array i.e. (2,4)

Boolean indexing

- Let us consider an array of 4 names with duplicates as follows:
- names = np.array(['Deepak', 'Lata', 'Shyamal', 'Lata'])
- Also consider a 4 X 3 array (arr) as follows:

```
arr = np.array([[ 0,  1,  2],  
                [-3, -4,  6],  
                [-8,  9, -1],  
                [7, -5,  0]])
```

- ba = (names == 'Lata') will produce a Boolean Array [False, True, False, True]
- The Boolean array can be passed when indexing an array (arr)
- print(arr[names == 'Lata']) will display [[-3 -4 6],
[7 -5 0]]
- The Boolean array must be of the same length as the axis it is indexing
- One can even mix and match Boolean arrays with slices and integers. Example:
- print(arr[names == 'Shyamal', 1:]) will display [[9 -1]]
- print(arr[names != 'Lata', 2]) will display [2 -1]

Boolean indexing

- Boolean arithmetic operators & (for AND) or | (for OR) may be used
- The Python keywords ‘and’ / ‘or’ do not work with Boolean arrays
- `mask = (names == 'Deepak') | (names == 'Shyamal')`
- `mask` will be a Boolean array [True, False, True, False]
- `print(arr[mask])` will display
[[0 1 2],
 [-8 9 -1]]
- Boolean Indexing always creates a copy of the array, even when the returned array is unchanged
- Setting values with Boolean arrays works in a common-sense way.
- `arr[arr < 0] = 0` will produce the array
[[0 1 2],
 [0 0 6],
 [0 9 0],
 [7 0 0]]

Fancy indexing

- Fancy indexing is a technique to describe indexing using integer arrays
- Fancy indexing , unlike slicing, always copies the data into a new array
- Let us consider a 4 X 3 array (arr) as follows:

```
([[ 0  1  2],  
 [-3 -4  6],  
 [-8  9 -1],  
 [7  -5  0]])
```

- To select out a subset of rows in a particular order, one can pass an ndarray of integers, specifying the desired order. Example:
- `print(arr[[2, 0, 3]])` will display

```
([[-8  9 -1],  
 [ 0  1  2],  
 [7  -5  0]])
```

-
- `print(arr[[-1,0, -3]])` will display

```
([[7  -5  0],  
 [ 0  1  2],  
 [-3 -4  6]])
```

Fancy indexing

- Passing multiple index array selects a 1-D array of elements corresponding to each tuple of indices:
- Let us consider the same 4 X 3 array (arr) as follows:

```
([[ 0  1  2],  
 [-3 -4  6],  
 [-8  9 -1],  
 [7  -5  0]])
```

- `print(arr[[2, 0, 3], [1, 2, 0]])` will display [9 2 7]
- Here, the elements (2,1), (0,2), (3,0) were selected

- `print(arr[[2, 0, 3]][:, [1, 2, 0]])` will display [[9 -1 -8], [1 2 0], [-5 0 7]]

Matrices in NumPy

- import numpy as np
 - a = np.matrix([[2,3,7], [1,-1,2]])
 - m = np.matrix('1 2 3; 4 5 6; 7 8 9')
 - np.diagonal(m) => Diagonal elements of m => [1, 5, 9]
 - np.max(a) => Maximum element of a => 7
 - np.min(a) => Minimum element of a => -1
 - np.sum(a) => Sum of all elements of a => 14
 - np.mean(m) => Average value of elements of m => 5.0
 - np.prod(m,0) => Column-wise product of all elements of m=> [28, 80,162]
 - np.prod(m,1) => Row-wise product of all elements of m => [[6] [120] [504]]

Matrices in NumPy

- import numpy as np
- m = np.matrix('2 1 8; 0 5 6; 7 9 3')
- n = np.matrix('1 -1 5; 4 -3 2; 6 7 -8')
- Matrix addition or elementwise addition ($p = m + n$)
- np.add(m,n) => [[3, 0, 13], [4, 2, 8], [13, 16, -5]]
- Matrix subtraction or elementwise subtraction ($p = m - n$)
- np.subtract(m,n) => [[1, 2, 3], [-4, 8, 4], [1, 2, 11]]
- Each element of m will be divided by the corresponding element of n ($p = m / n$)
- np.divide(m,n) => [[2.0, -1.0, 1.6], [0, -1.67, 3.0], [1.17, 1.2857, -0.375]]
- Elementwise remainder ($p = m \% n$)
- np.remainder(m,n) => [[0, 0, 3], [0, -1, 0], [1, 2, -5]]
- Elementwise multiplication
- np.multiply(m,n) => [[2, -1, 40], [0, -15, 12], [42, 63, -24]]
- Matrix multiplication as per rule of matrix algebra
- p =m * n => [[54, 51, -52], [56, 27, -38], [61, -13, 29]]

Matrices in NumPy

- `import numpy as np`
- `m = np.matrix('2 1 8; 0 5 6; 7 9 3')`
- `n = np.matrix('1 -1 5; 4 -3 2; 6 7 -8')`
- => Sort the elements in each row of m in ascending order
- `np.sort(m)` => `[[1, 2, 8], [0, 5, 6], [3, 7, 9]]`
- => Sort the elements in each column of m in ascending order
- `np.sort(m, axis=0)` => `[[0, 1, 3], [2, 5, 6], [7, 9, 8]]`
- => Transpose of the matrix m
- `np.transpose(m)` => `[[2, 0, 7], [1, 5, 9], [8, 6, 3]]`

Universal functions (*ufunc*)

- Universal functions in NumPy library are basically mathematical functions that performs elementwise operations on data in ndarrays. For example:
 - functions for arithmetic operations
 - standard trigonometric functions
 - handling complex numbers
 - statistical functions, etc.
- Universal functions have various characteristics which are as follows-
 - These functions operates on N-dimensional array (ndarray)
 - It performs fast element-wise array operations.
 - It supports various features like array broadcasting, type casting etc.
 - Some universal functions are called automatically when the corresponding arithmetic operator is used on arrays
 - For addition of two arrays (element-wise addition, using '+' operator), np.add() is called internally.
- Universal functions are instances of the numpy.ufunc class

Universal functions (Arithmetic)

- `import numpy as np`
- `arr = np.arange(5)` # arr contains [0 1 2 3 4] in ascending order
- `sqrtarr = np.sqrt(arr)` # finds the square root of the elements
- # sqrtarr contains [0. 1. 1.41421356 1.73205081 2.]
- `print(np.exp(arr))` # displays the values of e to the power 0 to 4
- # displays [1. 2.71828183 7.3890561 20.08553692 54.59815003]
- `x = np.array([2, 1, 8, 0, -1, 6, 7, 9, 0])`
- `y = np.array([1, -1, 5, 4, -3, 2, 6, 7, -8])`
- `print(np.maximum(x,y))` # displays the element-wise maximum
[2 1 8 4 -1 6 7 9 0]
- # ufunc: modf returns 2 arrays – one containing the fractional parts and the other containing the integral parts of the elements of a floating-point array
- `print(np.modf(sqrtarr))` # displays the following arrays
- `(array([0. , 0. , 0.41421356, 0.73205081, 0.]), array([0., 1., 1., 1., 2.]))`

Universal functions (Trigonometric)

- import numpy as np
- degrees = np.array([0,30,45,60,90]) # input angles in degrees
- radians = np.deg2rad(degrees) # convert degrees to radians
- [0. 0.52359878 0.78539816 1.04719755 1.57079633]
- sine_values =np.sin(radians) # find sine values
- [0. 0.5 0.70710678 0.8660254 1.]
- sinh_values = np.sinh(radians) # find sine hyperbolic values
- [0. 0.54785347 0.86867096 1.24936705 2.3012989]
- angle_radian = np.arcsin(sine_values) # find arc sine values
- [0. 0.52359878 0.78539816 1.04719755 1.57079633]
- angle_degree = np.rad2deg(angle_radian) # convert radian – degrees
- hypotenuse = np.hypot(3,4) # find length of hypotenuse (5.0)

Universal functions (Statistical)

- `import numpy as np`
- `weight = np.array([50.7, 52.5, 50, 58, 55.63, 73.25, 49.5, 45])`
- `np.amin(weight)` # Minimum value -> 45.0
- `np.amax(weight)` # Maximum value -> 73.25
- `np.ptp(weight)` # Peak-to-Peak = Range = $\max_v - \min_v$ -> 28.25
- `np.percentile(weight, 60)` # Weight below which 60% students fall (53.126)
- `np.mean(weight)` # Mean weight -> 54.3225
- `np.average(weight)` # Average -> 54.3225
- `np.median(weight)` # Median weight -> 51.6
- `np.var(weight)` # Variance -> 64.8471
- `np.std(weight)` # Standard deviation -> 8.05277

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Processing with NumPy
Lecture 21: 14/12/2021

Dr. Debranjan Sarkar

Universal functions (bit-twiddling)

- Import numpy as np
- arr1 = np.array([1 2 3 4])
- arr2 = np.array([5 6 7 8])
- num = 2
- np.bitwise_and(arr1, arr2) => bitwise AND operation => [1 2 3 0]
- np.bitwise_or(arr1, arr2) => bitwise OR operation => [5 6 7 12]
- np.bitwise_xor(arr1, arr2) => bitwise XOR operation => [4 4 4 12]
- np.invert(arr1) => bitwise NOT operation => [-2 -3 -4 -5]
- np.left_shift(arr1, num) => Left shift by num bits => [4 8 12 16]
- np.right_shift(arr1, num) => Right shift by num bits => [0 0 0 1]

More universal functions

- Unary universal functions
 - abs, square, log, log10, log2
 - ceil (smallest integer \geq each element) $\text{ceil}(2.3) \Rightarrow 3$
 - floor (largest integer \leq each element) $\text{floor}(2.3) \Rightarrow 2$
- Binary universal functions
 - power (raise elements in the 1st array to the powers indicated in the 2nd array)
 - copysign (copy sign of the values in the 2nd argument to the values in the 1st argument)
 - greater, greater_equal, less, less_equal, equal, not_equal (perform elementwise comparison, yielding Boolean array)
 - logical_and, logical_or, logical_xor (compute elementwise truth value of logical operation)

Aggregation

- Aggregations (or reductions) viz. sum, mean, standard deviation (std), variance (var) can be used by calling the array instance method or NumPy functions
- `import numpy as np`
- `arr = np.random.rand(15).reshape(3,5)`
- # create an array of 15 random numbers in shape (3,5) as follows:
[[0.33683763 0.00540638 0.29065752 0.29728181 0.30765457]
[0.45301324 0.98834843 0.64088087 0.36915614 0.15532608]
[0.32712496 0.83676363 0.23829175 0.1612305 0.41694673]]
- `print(arr.sum())` => Sum of all elements of arr 5.824920231190837
- `print(arr.sum(0))` => Sum of all the elements in each column
[1.11697583 1.83051844 1.16983014 0.82766845 0.87992738]

Aggregation

- import numpy as np
- arr = np.random.rand(15).reshape(3,5)
- # create an array of 15 random numbers in shape (3,5)
- print(arr.mean()) => Average value of elements of arr
0.38832801541272244
- print(np.mean(arr)) => Average value of elements of arr
0.38832801541272244
- print(arr.mean(axis = 1)) => Average values of all the elements in each row
[0.24756758 0.52134495 0.39607151]
- print(arr.std()) => Standard Deviation
0.24980124948304264
- print(arr.var()) => Variance
0.06240066424328931

Aggregation

- Methods like `cumsum` and `cumprod` do not aggregate but produce an array of the intermediate results
- `import numpy as np`
- `print(np.cumsum(0))` # Column-wise cumulative sum
[[0.33683763 0.00540638 0.29065752 0.29728181 0.30765457]
[0.78985087 0.99375481 0.93153839 0.66643795 0.46298065]
[1.11697583 1.83051844 1.16983014 0.82766845 0.87992738]]
- `print(arr.cumprod(1))` # Row-wise cumulative product
[[3.36837627e-01 1.82107336e-03 5.29308670e-04 1.57353841e-04 4.84106285e-05]
[4.53013241e-01 4.47734924e-01 2.86944747e-01 1.05927415e-01 1.64532897e-02]
[3.27124962e-01 2.73726269e-01 6.52267114e-02 1.05165351e-02 4.38483494e-03]]

Computation on Arrays: Broadcasting

- Operations between differently sized arrays is called broadcasting
- Broadcasting describes how arithmetic works between arrays of different shapes
- Combining a scalar (array with 0 dimension) value with an array:
 - `arrin = np.arange(5)` # Create an array [0,1,2,3,4]
 - `arrout = arrin + 7` # Add each element of arrin by 7
 - This stretches or duplicates the value 7 into the array [7,7,7,7,7] and adds with arrin
 - The scalar value 7 is said to have been broadcast to all the elements of the arrin
 - In NumPy, this duplication of values does not actually take place, but this is a useful mental model about broadcasting
- Another example to add a 1-D array with a 2-D array:
 - `x = np.ones((3,3))` # create a 3 X 3 matrix [[1. 1. 1.], [1. 1. 1.], [1. 1. 1.]]
 - `y = np.array([0,1,2])` # create a 1 X 3 array [0, 1, 2]
 - `z = x + y` # The 1-D array (y) is broadcast to match the shape of x
[[0, 1, 2],[0, 1, 2],[0, 1, 2]]
 - `print(z)` # will display [[1. 2. 3.], [1. 2. 3.], [1. 2. 3.]]

Broadcasting

- Example to show broadcasting of both the arrays

```
m = np.matrix('0 1 2') # create a row matrix m => [0 1 2]
```

```
n = np.matrix('0; 1; 2') # create a column matrix n => [[0],  
[1],  
[2]]
```

- $p = m + n$

- # Both m and n are broadcast to a 3×3 matrix each

```
m => [[0 1 2],  
       [0 1 2] ,  
       [0 1 2]]
```

```
n => [[0 0 0],  
       [1 1 1],  
       [2 2 2]]
```

- `print(p)`

- Displays the 3×3 matrix

```
[[0. 1. 2.],  
 [1. 2. 3.],  
 [2. 3. 4.]]
```

Rules of broadcasting

- Rule 1: If the number of dimensions of the two arrays are different, the shape of the array with fewer dimensions is padded with 1's on its left (leading) side
- Rule 2: If the shapes of the two arrays do not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape
- Rule 3: If in any dimension, the sizes disagree and not equal to 1, an error is raised

Broadcasting: Example 1

- Adding a 2-D array with a 1-D array
- `x = np.ones((2,3))` # create a 2 X 3 array with all elements as 1.0
- `y = np.arange(3)` # create a 1-D array of 3 elements [0,1,2]
- `z = x + y`
- Shape of x is (2,3) and shape of y is (3,)
- y has fewer dimensions. We pad with 1 to the left of its shape [Rule 1]
- So shape of y becomes (1,3) and that of x remains (2,3)
- As the first dimension of x does not match with that of y, we stretch the first dimension of y to 2 [Rule 2]
- Now both the shapes are (2,3)
- So `x = [[1. 1. 1.] [1. 1. 1.]]` and `y = [[0 1 2] [0 1 2]]`
- `z = [[1. 2. 3.] [1. 2. 3.]]`

Broadcasting: Example 2

- An example where two arrays are not compatible
- `x = np.ones((3,2))` # create a 3 X 2 array with all elements as 1.0
- `y = np.arange(3)` # create a 1-D array of 3 elements [0,1,2]
- `z = x + y`
- Shape of x is (3,2) and shape of y is (3,)
- As y has fewer dimensions, we pad with 1 to the left of its dimension [Rule 1]
- So shape of y becomes (1,3) and that of x remains (3,2)
- As the first dimension of y is 1 and it does not match with that of x, we stretch the first dimension of y to 3 [Rule 2]
- The shape of y becomes (3,3) and that of x remains (3,2)
- The final shapes of x and y do not match. So these two arrays are incompatible [Rule 3]
- If we run the above code, it generates an exception
`ValueError: operands could not be broadcast together with shapes (3,2) (3,)`

Broadcasting: Example 3

- Demeaning of each column of an array can be done by subtracting the column means from each column
- `arr = np.matrix('2,5,7; -2,3,0; 4,2,1; 6,0,8')` # arr has a shape (4,3)
- `colmean = arr.mean(0)` # colmean = [[2.5 2.5 4.]] & shape (3,)
- `demeancol = arr - colmean` # Demeaning of each column
- Apply rules for broadcasting
- Shape of arr (4,3)
- Shape of colmean (3,) Rule1->(1,3) Rule2->(4,3)
- `demeancol =` `[-0.5 2.5 3.]` `colmean =` `[[2.5 2.5 4.]`
`[-4.5 0.5 -4.]` `[2.5 2.5 4.]`
`[1.5 -0.5 -3.]` `[2.5 2.5 4.]`
`[3.5 -2.5 4.]]` `[2.5 2.5 4.]]`
- `print(demeancol.mean(0))` # will display array [[0. 0. 0.]]

Broadcasting: Example 4

- Demeaning of each row of an array can be done by subtracting the row means from each row
- `arr = np.matrix('2,5,7; -2,3,0; 4,2,1; 6,0,8')` # arr has a shape (4,3)
- `rowmean = arr.mean(1)` # rowmean has a shape (4,)
- `demeanrow = arr - rowmean` # Demeaning of each row
- Apply rules for broadcasting
- Shape of rowmean (4,) Rule1-> (1,4) Rule2-> (4,4)
- As per Rule3 of broadcasting, the two arrays (arr and rowmean) are incompatible
- So we have to reshape the rowmean:
`rowmean.reshape((4,1))`
- Shape of rowmean (4,1) Rule2-> (4,3) -> compatible
- `print(demeanrow.mean(1))` # will display array [[0. 0. 0.]]

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Processing with NumPy
Lecture 22: 03/01/2022

Dr. Debranjan Sarkar

Expressing conditional logic as array operations

- Let us consider three arrays as follows:
- `xarr = np.array([1.0, 1.5, 2.0, 2.5, 3.0])`
- `yarr = np.array([-1.0, -1.5, -2.0, -2.5, -3.0])`
- `cond = np.array([True, False, True, True, False])`
- We want to take a value from `xarr` if the corresponding element in `cond` is `True`, otherwise we take the value from `yarr`
- A list comprehension for this is as follows:
- `outarr = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]`
- `print(outarr)` will display [1.0, -1.5, 2.0, 2.5, -3.0]

Expressing conditional logic as array operations

- The previous method has the following problems:
 - Not very fast for large arrays (because all the work is done in pure Python)
 - Does not work with multi-dimensional arrays
- So we use np.where as follows:
- `outarr = np.where(cond, xarr, yarr)` # Same action as above
- The second and the third arguments of np.where may not be arrays.
- One or both of them can be scalars. Example:
- `arr = np.random.randn(4,4)`
- Create a 4 X 4 matrix of random numbers (+ or -)
- `np.where(arr > 0, 3, -3)` # Set positive values to 3 otherwise -3
- `np.where(arr > 0, 3, arr)` # Set positive values to 3 otherwise no change

Expressing conditional logic as array operations

- ‘where’ can be used to express more complicated logic
- Consider two Boolean arrays cond1 and cond2
- cond1 = [True, False, True, True, False]
- cond2 = [True, True, False, True, False]
- n = 5
- result = []
- for i in range(n):

```
    if cond1[i] and cond2[i]:  
        result.append(0)  
    elif cond1[i]:  
        result.append(1)  
    elif cond2[i]:  
        result.append(2)  
    else:  
        result.append(3)
```

- print(result)

will display

[0, 2, 1, 0, 3]

Expressing conditional logic as array operations

- This ‘for’ loop can be converted to a nested where expression
- `np.where(cond1 & cond2, 0,
 np.where(cond1, 1,
 np.where(cond2, 2, 3)))`

Methods for Boolean arrays

- Boolean values are 1 (True) and 0 (False)
- So, sum means the total number of True values in a Boolean array
- `arr = np.random.randn(50)`
- `print((arr > 0).sum())` will display the number of positive values
- The method ‘any’ is to inform whether there is at least one True value in an array
- `arr = [False, True, False, False, False]`
- `print(arr.any())` will display True
- The method ‘all’ is to inform whether all the values in an array are True
- `arr = [False, True, False, False, False]`
- `print(arr.all())` will display False
- These methods also work with non-Boolean arrays where non-zero elements are True

Advantages of NumPy : Speed

- NumPy supports vectorized operations
- When compared with the operations carried out on Python lists, Universal functions (ufunc) in NumPy run faster as the Array operations are carried out in C
- To compare speed, we use ‘timeit’ module to measure the execution time of a code
- Comparing the processing speed of a list and an array using an addition operation
- `x = range(1000)` # Create a list
- `=====`
- `y = np.array(x)` # Create a NumPy array
- `np.sum(y)`
- It was observed that array works faster compared with the lists

How to use the method timeit (List)

- `import timeit` # Required module to be imported
- Code snippet to be executed only once
- `mysetup = ""`
- `import numpy as np` # NumPy module is to be imported
- `x = range(1000)` # Create a list
- `"`
- Code snippet whose execution time is to be measured
- `mycode = "`
- `sum(x)`
- `"`
- `print(timeit.timeit(setup = mysetup, stmt = mycode, number = 100000))`
- This displayed 2.015 sec That means the execution time for a list is 20.15 μ sec

How to use the method timeit (Array)

- `import timeit` # Required module to be imported
- Code snippet to be executed only once
- `mysetup = ""`
- `import numpy as np` # NumPy module is to be imported
- `x = range(1000)` # Create a list
- `y = np.array(x)`
- `"`
- Code snippet whose execution time is to be measured
- `mycode = ""`
- `np.sum(y)`
- `"`
- `print(timeit.timeit(setup = mysetup, stmt = mycode, number = 100000))`
- This displayed 0.466 sec That means the execution time for an array is 4.66 μ sec
- The execution time for an array is approx. 4.5 times faster compared to that for a list

Advantages of NumPy : Storage space

- Use the same examples x and y to compare the memory used at run time
- `getsizeof()` is a built in function which returns the size of the object in bytes
- `Syntax: sys.getsizeof(object)` # we have to import the package `sys`
- Used for system specific parameters
- Size of the list can be found by multiplying the size of an individual element with the number of elements in the list
- `sys.getsizeof(1) * len(x)` => 14000
- Size of an array can be found by multiplying the size of an individual element with the number of elements in the array
- `itemsize` returns the size of one element of a NumPy array
- `Syntax: numpy.ndarray.itemsize`
- `y.itemsize * y.size` => 4000 => 3.5 times less compared to a list

Slowness of loops

- NumPy arrays are faster than the ‘for’ looping over lists in Python, because:
- [1] Numpy is written mostly in C language, and the whole data structure is vectorized
- [2] NumPy takes advantage of parallelism (Single Instruction Multiple Data (SIMD))
 - Traditional ‘for’ loop can’t make use of it
- [3] Python is dynamically typed.
 - That means, user don’t have to specify data type of a variable
 - But every time Python uses a variable, it has to check the data type
 - This makes it slower
- [4] NumPy arrays are densely packed arrays of homogeneous numerically packed data type
- [5] Python lists, by contrast, are arrays of pointers to objects
 - Even when all of them are of the same type, memory is dynamically allocated
 - So the benefit of locality of reference is lost

Bibliography

- “Python for Data Analysis”, Wes McKinney, (O’Reilly)
- NPTEL Lecture by Prof. Madhavan Mukund on “Programming, Data Structure and Algorithms using Python”
- “Exploring Python”, Timothy A Budd, (Tata McGraw-Hill Education Pvt. Ltd., New Delhi)
- “Core Python Programming”, R. Nageswara Rao, (dreamtech press)
- “Introduction to Computing & Problem Solving with Python”, Jeeva Jose & P. Sojan Lal, (Khanna Book Publishing)
- “Effective Python”, Brett Slatkin, (Pearson)
- “Data Structures and Algorithms in Python”, Goodrich, Tamassia & Goldwasser, (Wiley Student Edition)
- “Introduction to Machine Learning with Python”, Muller & Guido, (O’Reilly)

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Data Manipulations with Pandas
Lecture 23: 04/01/2022

Dr. Debranjan Sarkar

What is ‘pandas’?

- ‘pandas’ is a data analysis and manipulation tool, built on top of the Python programming language.
- It is popular mainly because it is fast, powerful, flexible, easy-to-use and open source
- It offers data structures and operations for manipulating numerical tables and time series.
- Pandas stands for “Python Data Analysis Library ”.
- The library’s name derives from panel data, a common term for multidimensional data sets encountered in statistics and econometrics.
- Pandas can be imported as “import pandas as pd”

Pandas Data Structure: Series

- Two important data structures in pandas: Series and DataFrame
- A series is a 1-D array-like object containing
 - An array of data, and
 - An associated array of data labels, called index
- `obj = Series([2,7,-3,9])`
- `print(obj)` will display

0	2
1	7
2	-3
3	9
- The left column represents the index (by default 0, 1, 2, ...) and the right column represents the values
- `print(obj.values)` will display `array([2, 7, -3, 9])`
- `print(obj.index)` will display `RangIndex(start=0, stop=4, step=1)`

Pandas Data Structure: Series

- Sometimes it is required to create a Series with an index for each data point
- `obj = Series([2,7,-3,9], index = ['d', 'b', 'a', 'c'])`
- `print(obj)` will display

d	2
b	7
a	-3
c	9
- `print(obj.index)` will display `Index(['d', 'b', 'a', 'c'], dtype='object')`
- One can use values in the index when selecting single values or a set of values
- `print(obj['a'])` will display -3
- `obj['d'] = 6` will replace 2 by 6
- `print(obj[['c', 'd', 'a']])` will display

c	9
d	6
a	-3

Pandas Data Structure: Series

- Filtering with Boolean array, scalar multiplication, or applying math functions will preserve the index-value link

- `print(obj)` will display

d	6
b	7
a	-3
c	9

- `obj1 = obj[obj > 6]`

- `print(obj1)` will display

b	7
c	9

- `obj2 = obj1 * 3`

- `print(obj2)` will display

b	21
c	27

- `obj3 = np.exp(obj1)`

- `print(obj3)` will display

b	1096.633158
c	8103.083928

Pandas Data Structure: Series

- A Series can be thought of as a fixed-length, ordered dictionary
- Because it is a mapping of index values to data values
- Many functions that expect a dictionary, can be used for a Series
- ‘b’ in obj1 => True ‘d’ in obj1 => False
- A Series can be created from a dictionary as follows:
- `dict = {'India': 1326, 'USA': 330, 'China': 1394, 'Russia': 141}`
- # Population of some countries in Millions
- `obj1 = Series(dict)`
- `print(obj1)` will display

India	1326
USA	330
China	1394
Russia	141

Pandas Data Structure: Series

- Let, popul = ['Bangladesh', 'India', 'USA', 'China']
- Note that 'Bangladesh' is not a key in the dictionary
- obj2 = Series(dict, index = popul)
- print(obj2) will display

Bangladesh	NaN	# Not a Number
India	1326.0	
USA	330.0	
China	1394.0	

- 'isnull' or 'notnull' functions are used to detect NaN
- print(pd.isnull(obj2)) will display

Bangladesh	True
India	False
USA	False
China	False

Pandas Data Structure: Series

- `print(pd.notnull(obj2))` will display

Bangladesh	False
India	True
USA	True
China	True

- Series also has ‘isnull’ and ‘notnull’ as instance methods

- `obj2.isnull()` will display

Bangladesh	True
India	False
USA	False
China	False

- `print(obj1 + obj2)` will display

Bangladesh	NaN
China	2788.0
India	2652.0
Russia	NaN
USA	660.0

Note that this is in alphabetical order of the index

Pandas Data Structure: Series

- Both the Series object itself and its index have a ‘name’ attribute
- obj2.name = ‘population’
- obj2.index.name = ‘Country’
- print(obj2) will display

```
Country
Bangladesh    NaN
India          1326.0
USA            330.0
China          1394.0
Name: population, dtype: float64
```

Pandas Data Structure: Series

- The index of a Series can be altered in-place by assignment

- `obj = Series([92,87,93,89], index = ['D', 'B', 'A', 'C'])`

- `print(obj)` will display

	D	92
	B	87
	A	93
	C	89

- `obj.index = ['Deepak', 'Biswanath', 'Abhishek', 'Chandrima']`

- `print(obj)` will display

Deepak	92
Biswanath	87
Abhishek	93
Chandrima	89

Pandas Data Structure: DataFrame

- A DataFrame is a tabular, spreadsheet-like data structure
- It contains an ordered collection of columns, each of which can be a different value type (numeric, Boolean, string etc.)
- It has row index and column index
- A DataFrame may be thought of as a dictionary of Series
- Row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically
- The data is stored as one or more two-dimensional blocks rather than a list, dictionary, or some other collection of one-dimensional arrays
- Higher-dimensional data can easily be represented in a tabular format using hierarchical indexing

Pandas Data Structure: DataFrame

- Construction of DataFrame from a dictionary of equal-length lists or NumPy arrays
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'IX', 'X', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame = DataFrame(data)`
- `print(frame)` will display (with index assigned automatically, as in Series)

	Subjects	Class	Marks
0	Phy. Sc.	IX	92
1	Maths.	IX	95
2	Phy. Sc.	X	94
3	Maths.	X	99

Pandas Data Structure: DataFrame

- If a sequence of columns is specified, the columns of the DataFrame will be arranged accordingly
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'X', 'IX', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame = DataFrame(data, columns = ['Class', 'Subjects', 'Marks'])`
- `print(frame)` will display as per the specified arrangement of the columns

	Class	Subjects	Marks
0	IX	Phy. Sc.	92
1	X	Maths.	95
2	IX	Phy. Sc.	94
3	X	Maths.	99

Pandas Data Structure: DataFrame

- If a column that is not contained in data, is passed, it will appear as NaN in the result
- ```
data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'],
 'Class': ['IX', 'IX', 'X', 'X'],
 'Marks': [92, 95, 94, 99]}
```
- ```
frame2 = DataFrame(data, columns = ['Class', 'Subjects', 'Marks', 'Highest_Marks'],
                     index = ['One', 'Two', 'Three', 'Four'])
```
- `print(frame2)` will display as follows:

	Class	Subjects	Marks	Highest_Marks
One	IX	Phy. Sc.	92	NaN
Two	IX	Maths.	95	NaN
Three	X	Phy. Sc.	94	NaN
Four	X	Maths.	99	NaN

- `print(frame2.columns)` will display

```
Index(['Class', 'Subjects', 'Marks', 'Highest_Marks'], dtype='object')
```

Pandas Data Structure: DataFrame

- A column in a DataFrame can be retrieved as a Series either by dictionary-like notation or by attribute
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'IX', 'X', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame2 = DataFrame(data, columns = ['Class', 'Subjects', 'Marks', 'Highest_Marks'], index = ['One', 'Two', 'Three', 'Four'])`
- `print(frame2['Subjects'])` will display

One	Phy. Sc.
Two	Maths.
Three	Phy. Sc.
Four	Maths.

Name: Subjects, dtype: object

- `print(frame2.Marks)` will display

One	92
Two	95
Three	94
Four	99

Name: Marks, dtype: int64

Pandas Data Structure: DataFrame

- Rows in a DataFrame can be retrieved by position or name by different methods
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'IX', 'X', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame2 = DataFrame(data, columns = ['Class', 'Subjects', 'Marks', 'Highest_Marks'], index = ['One', 'Two', 'Three', 'Four'])`
- `print(frame2)` will display as follows:

	Class	Subjects	Marks	Highest_Marks
One	IX	Phy. Sc.	92	NaN
Two	IX	Maths.	95	NaN
Three	X	Phy. Sc.	94	NaN
Four	X	Maths.	99	NaN

- `print(frame2.loc['Three'])` will display as follows:

```
Class          X  
Subjects      Phy. Sc.  
Marks         94  
Highest_Marks NaN  
Name: Three, dtype: object
```

Pandas Data Structure: DataFrame

- Columns can be modified by assignment
- For example, the ‘Highest_Marks’ column can be assigned a scalar value or an array of values
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'IX', 'X', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame2 = DataFrame(data, columns = ['Class', 'Subjects', 'Marks', 'Highest_Marks'], index = ['One', 'Two', 'Three', 'Four'])`

- `frame2['Highest_Marks'] = 99`

- `print(frame2)` will display

	Class	Subjects	Marks	Highest_Marks
One	IX	Phy. Sc.	92	99
Two	IX	Maths.	95	99
Three	X	Phy. Sc.	94	99
Four	X	Maths.	99	99

- `frame2['Highest_Marks'] = [95, 100, 96, 99]`

- `print(frame2)` will display

	Class	Subjects	Marks	Highest_Marks
One	IX	Phy. Sc.	92	95
Two	IX	Maths.	95	100
Three	X	Phy. Sc.	94	96
Four	X	Maths.	99	99

Pandas Data Structure: DataFrame

- When assigning a list or an array to a column, the length of the value must be equal to the length of the DataFrame
- While assigning a Series, its index should be exactly same as the index of the DataFrame, inserting missing values in any holes
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'IX', 'X', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame2 = DataFrame(data, columns = ['Class', 'Subjects', 'Marks', 'Highest_Marks'], index = ['One', 'Two', 'Three', 'Four'])`
- `val = Series([95, 96, 99], index = ['One', 'Three', 'Four'])`
- `frame2['Highest_Marks'] = val`
- `print(frame2)` will display

	Class	Subjects	Marks	Highest_Marks
One	IX	Phy. Sc.	92	95.0
Two	IX	Maths.	95	NaN
Three	X	Phy. Sc.	94	96.0
Four	X	Maths.	99	99.0

Pandas Data Structure: DataFrame

- Assigning a non-existent column will create a new column
- `data = {'Subjects': ['Phy. Sc.', 'Maths.', 'Phy. Sc.', 'Maths.'], 'Class': ['IX', 'IX', 'X', 'X'], 'Marks': [92, 95, 94, 99]}`
- `frame2 = DataFrame(data, columns = ['Class', 'Subjects', 'Marks', 'Highest_Marks'], index = ['One', 'Two', 'Three', 'Four'])`
- `frame2['Excellent'] = frame2.Highest_Marks - frame2.Marks < 5`
- `print(frame2)` will display

	Class	Subjects	Marks	Highest_Marks	Excellent
One	IX	Phy. Sc.	92	95.0	True
Two	IX	Maths.	95	NaN	False
Three	X	Phy. Sc.	94	96.0	True
Four	X	Maths.	99	99.0	True

- The 'del' keyword will delete columns (like dictionary)
- `del frame2['Excellent']`
- `print(frame2.columns)` will display

`Index(['Class', 'Subjects', 'Marks', 'Highest_Marks'], dtype='object')`

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Data Manipulations with Pandas
Lecture 24: 05/01/2022

Dr. Debranjan Sarkar

Pandas Data Structure: DataFrame

- Another common form of data is a nested dictionary
 - `marks = {'Phy. Sc.': {'X': 94, 'IX': 92}, 'Maths.': {'IX': 95, 'X': 99}}`
 - When passed to DataFrame, it will interpret the outer dictionary keys as the columns and the inner keys as the row indices
 - `frame3 = DataFrame(marks)`
 - `print(frame3)` will display
- | | Phy. Sc. | Maths. |
|----|----------|--------|
| X | 94 | 99 |
| IX | 92 | 95 |
- The keys in the inner dictionaries are arranged to form the index in the result

Pandas Data Structure: DataFrame

- However, if an explicit index is specified, it will follow that index
- `print(DataFrame(marks, index = ['IX', 'X', 'XI']))` will display

	Phy. Sc.	Maths.
IX	92.0	95.0
X	94.0	99.0
XI	NaN	NaN

- The result can also be transposed
- `print(frame3.T)` will display

	X	IX
Phy. Sc.	94	92
Maths.	99	95

- Dictionaries of Series are treated much in the same way
- `mdata = {'Maths.': frame3['Maths.'][:-1], 'Phy. Sc.': frame3['Phy. Sc.'][2:]}`
- `print(DataFrame(mdata))` will display

	Maths.	Phy. Sc.
IX	NaN	92
X	99.0	94

Pandas Data Structure: DataFrame

- If the index and columns of the DataFrame have their name attributes set, these will also be displayed
- `frame3.index.name = 'Class'; frame3.columns.name = 'Subject'`
- `print(frame3)` will display

	Subject	Phy. Sc.	Maths.
Class			
X		94	99
IX		92	95

- Like Series, the ‘values’ attribute returns the data contained in the DataFrame as a 2D ndarray
- `print(frame3.values)` will display
- If the columns of the DataFrame are of different dtypes, the dtype of the values array will be chosen such that all the columns can be accommodated
- `print(frame2.values)` will display

[[94 99]
[92 95]]

[['IX' 'Phy. Sc.' 92 95.0 True]
['IX' 'Maths.' 95 nan False]
['X' 'Phy. Sc.' 94 96.0 True]
['X' 'Maths.' 99 99.0 True]]

Index Objects

- Index objects in pandas hold the axis labels and other metadata (e.g. names of axis)
- Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index
- `obj = Series(range(4), index = ['a', 'b', 'c', 'd'])`
- `ind = obj.index`
- `print(ind)` will display `Index(['a', 'b', 'c', 'd'], dtype='object')`
- `print(ind[1:3])` will display `Index(['b', 'c'], dtype='object')`
- Index objects are immutable and cannot be modified by the user
- `ind[1] = 'e'` will give rise to the following exception
`TypeError: Index does not support mutable operations`

Index Objects

- As Index objects are immutable, these can be safely shared among data structures
- `ind = pd.Index(np.arange(4))`
- `print(ind)` will display `Int64Index([0, 1, 2, 3], dtype='int64')`
- `obj2 = Series([1.5, 2.5, 3.5, 4.5], index = ind)`
- `print(obj2)` will display

0	1.5
1	2.5
2	3.5
3	4.5

`dtype: float64`
- `print(obj2.index is ind)` will display True

Index Objects

- Index is normally like array, but it can also function as a fixed-size set
- `marks = {'Phy. Sc.': {'X': 94, 'IX': 92}, 'Maths.': {'IX': 95, 'X': 99}}`
- `frame3 = DataFrame(marks)`

- `print(frame3)` will print

	Phy. Sc.	Maths.
X	94	99
IX	92	95

- `print('Maths' in frame3.columns)`

will display True

- `print('XI' in frame3.index)`

will display False

Reindexing

- Reindexing is to create a new object with the data in conformity with a new index.
Example:
- `obj = Series([1.5, 3.0, -4.5, -6.0], index = ['d', 'b', 'c', 'a'])`
- `print(obj)` will display

	d	1.5
	b	3.0
	c	-4.5
	a	-6.0

- `obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])` # rearranges the data as per the new index
- `print(obj2)` will display

	a	-6.0
	b	3.0
	c	-4.5
	d	1.5
	e	NaN

- `obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value = 0)` # replaces NaN by the fill value 0.0

Reindexing

- The ‘method’ option allows to fill the values while reindexing, using a method viz. ffill, bfill etc.
- `obj3 = Series(['Red', 'Green', 'Blue'], index = [0, 2, 4])`
- `print(obj3.reindex(range(5), method = 'ffill'))` will display the following:

0	Red
1	Red
2	Green
3	Green
4	Blue

- `print(obj3.reindex(range(6), method = 'bfill'))` will display the following:

0	Red
1	Green
2	Green
3	Blue
4	Blue
5	NaN

Reindexing

- With DataFrame, reindex can alter the row (index), or columns or both. Example:
- frame = DataFrame(np.arange(12).reshape(3,4), index = ['A', 'C', 'D'], columns = ['Kolkata', 'Delhi', 'Mumbai', 'Chennai'])
- print(frame) will display

	Kolkata	Delhi	Mumbai	Chennai
A	0	1	2	3
C	4	5	6	7
D	8	9	10	11

- print(frame.reindex(['A', 'B', 'C', 'D'])) will display

	Kolkata	Delhi	Mumbai	Chennai
A	0.0	1.0	2.0	3.0
B	NaN	NaN	NaN	NaN
C	4.0	5.0	6.0	7.0
D	8.0	9.0	10.0	11.0

Reindexing

- The columns can be reindexed using the columns keyword
- cities = ['Kolkata', 'Bengaluru', 'Mumbai']
- print(frame.reindex(columns = cities)) will display

	Kolkata	Bengaluru	Mumbai
A	0	NaN	2
C	4	NaN	6
D	8	NaN	10

- Both rows and columns can be reindexed in one shot, and fill value can be provided
- print(frame.reindex(index = ['A', 'B', 'C', 'D'], columns = cities, fill_Value = 25)) will display:

	Kolkata	Bengaluru	Mumbai
A	0	25	2
B	25	25	25
C	4	25	6
D	8	25	10

Dropping entries from an axis

- The ‘drop’ method returns a new object with the indicated value(s) deleted from an axis
- `obj = Series(np.arange(5.), index = ['(1)', '(2)', '(3)', '(4)', '(5)'])`
- `obj1 = obj.drop(['(3)'])`
- `print(obj)` will display
 - (1) 0.0
 - (2) 1.0
 - (3) 2.0
 - (4) 3.0
 - (5) 4.0
- `print(obj.drop(['(4)', '(2)']))` will display
 - (1) 0.0
 - (3) 2.0
 - (5) 4.0

Dropping entries from an axis

- With DataFrame, index values can be deleted from either axis
 - frame = DataFrame(np.arange(12).reshape((3,4)), index = 'A', 'B', 'C'],
columns = ['Kolkata', 'Delhi', 'Mumbai', 'Chennai'])
 - print(frame) will display
- | | Kolkata | Delhi | Mumbai | Chennai |
|---|---------|-------|--------|---------|
| A | 0 | 1 | 2 | 3 |
| B | 4 | 5 | 6 | 7 |
| C | 8 | 9 | 10 | 11 |
- print(frame.drop(['B'])) will display
 - print(frame.drop(['Mumbai'], axis = 1)) will display
- | | Kolkata | Delhi | Mumbai | Chennai |
|---|---------|-------|--------|---------|
| A | 0 | 1 | 2 | 3 |
| C | 8 | 9 | 10 | 11 |

Indexing, Selection and Filtering

Indexing, Selection and Filtering

- For slicing with labels, both the startpoint and the endpoint are included
- `print(obj[‘(2)’:‘(4)’])` will display
 - (2) 1.0
 - (3) 2.0
 - (4) 3.0
- Setting, using these methods, is normal
- `obj[‘(2)’:‘(4)’] = 5.0`
- `print(obj)` will display
 - (1) 0.0
 - (2) 5.0
 - (3) 5.0
 - (4) 5.0
 - (5) 4.0

Indexing, Selection and Filtering

- Indexing into a DataFrame is to retrieve one or more columns either with a single value or sequence
- `frame = DataFrame(np.arange(12).reshape((3,4)), index = 'A', 'B', 'C'], columns = ['Kolkata', 'Delhi', 'Mumbai', 'Chennai'])`

- `print(frame)` will display

	Kolkata	Delhi	Mumbai	Chennai
A	0	1	2	3
B	4	5	6	7
C	8	9	10	11

- `print(frame[[['Kolkata', 'Chennai']]])` will display

	Kolkata	Chennai
A	0	3
B	4	7
C	8	11

Indexing, Selection and Filtering

- `print(frame[1:3])` will display

		Kolkata	Delhi	Mumbai	Chennai
	B	4	5	6	7
	C	8	9	10	11

- `print(frame[frame['Mumbai'] < 10])` will display

		Kolkata	Delhi	Mumbai	Chennai
	A	0	1	2	3
	B	4	5	6	7

- `print(frame < 10)` will display

		Kolkata	Delhi	Mumbai	Chennai
	A	True	True	True	True
	B	True	True	True	True

- `frame[frame < 10] = 0`

- `print(frame)` will display

		Kolkata	Delhi	Mumbai	Chennai
	A	0	0	0	0
	B	0	0	0	0

- This makes the DataFrame syntactically more like an ndarray

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Data Manipulations with Pandas
Lecture 25: 10/01/2022 (12:25 hrs.)

Dr. Debranjan Sarkar

Arithmetc and data alignment

- `s1 = Series([1.5, 2.0, -1.5, -2.0], index = ['A', 'B', 'C', 'D'])`
- `s2 = Series([1.5, -2.0, 1.5, -2.0, 2.5], index = ['A', 'C', 'D', 'E', 'F'])`

`print (s1)`

A 1.5

B 2.0

C -1.5

D -2.0

`print(s2)`

A 1.5

C -2.0

D 1.5

E -2.0

F 2.5

`print(s1 + s2)`

A 3.0

B NaN

C -3.5

D -0.5

E NaN

F NaN

Arithmetric and data alignment

- df1 = DataFrame(np.arange(9).reshape(3,3),
index = ['BOM', 'CCU', 'DEL'], columns = list('bcd'))
- df2= DataFrame(np.arange(12).reshape(4,3),
index = ['MAS', 'CCU','DEL', 'PAT'], columns = list('bde'))

	print(df1)	print(df2)	print(df1+df2)
	b c d	b d e	b c d e
BOM	0 1 2	MAS 0 1 2	BOM NaN NaN NaN NaN
CCU	3 4 5	CCU 3 4 5	CCU 6.0 NaN 9.0 NaN
DEL	6 7 8	DEL 6 7 8	DEL 12.0 NaN 15.0 NaN
		PAT 9 10 11	MAS NaN NaN NaN NaN
			PAT NaN NaN NaN NaN

Arithmetic methods with fill values

- df1 = DataFrame(np.arange(9).reshape(3,3), index = ['BOM', 'CCU','DEL'], columns = list('bcd'))
- df2= DataFrame(np.arange(12).reshape(4,3),
index = ['MAS', 'CCU','DEL', 'PAT'], columns = list('bde'))
- Using the add method on df1, with one argument as df2, fill_value may be passed as other argument

	df1	df2	print (df1.add(df2, fill_value = 0))
	b c d e	b c d e	b c d e
BOM	0 1 2 *	BOM * * * *	BOM 0.0 1.0 2.0 NaN
CCU	3 4 5 *	CCU 3 * 4 5	CCU 6.0 4.0 9.0 5.0
DEL	6 7 8 *	DEL 6 * 7 8	DEL 12.0 7.0 15.0 8.0
MAS	* * * *	MAS 0 * 1 2	MAS 0.0 NaN 1.0 2.0
PAT	* * * *	PAT 9 * 10 11	PAT 9.0 NaN 10.0 11.0

- If one of the elements of df1 or df2 is a number, the other NaN is filled with 0
- If both the elements of df1 and df2 are not numbers, the result is NaN

Arithmetic methods with fill values

- df1 = DataFrame(np.arange(9).reshape(3,3),
index = ['BOM', 'CCU', 'DEL'], columns = list('bcd'))
- df2= DataFrame(np.arange(12).reshape(4,3),
index = ['MAS', 'CCU', 'DEL', 'PAT'], columns = list('bde'))
- When reindexing a Series or a DataFrame, one can specify a different fill value
- print(df1.reindex(columns = df2.columns, fill_value = 0)) will display the following:

	df1			df2			df1.reindex(columns = df2.columns, fill_value = 0)		
	b	c	d	b	d	e	b	d	e
BOM	0	1	2	MAS	0	1	2	0	2
CCU	3	4	5	CCU	3	4	5	3	5
DEL	6	7	8	DEL	6	7	8	6	8
				PAT	9	10	11		0

Operations between DataFrame and Series

- Let us recall the add operation of a 2D array and one of its rows
- `arr = np.arange(12.).reshape((3,4))`
- To add `arr[1]` with `arr`, broadcasting is to be done

```
print(arr)
```

```
[[ 0.  1.  2.  3.]
```

```
[ 4.  5.  6.  7.]
```

```
[ 8.  9. 10. 11.]]
```

```
print(arr[1])
```

```
[4.  5.  6.  7.]
```

```
print(arr + arr[1])
```

```
[[ 4.  6.  8. 10.]
```

```
[ 8. 10. 12. 14.]
```

```
[12. 14. 16. 18.]]
```

- Operations between a DataFrame and a Series are similar. Example:
- `df1 = DataFrame(np.arange(9).reshape((3,3)), index = ['BOM', 'CCU', 'DEL'], columns = list('bcd'))`
- `s1 = df1.loc['CCU']`

```
print(df1)
```

```
b c d
```

```
BOM  0 1 2
```

```
CCU  3 4 5
```

```
DEL  6 7 8
```

```
print(s1)
```

```
b 3
```

```
c 4
```

```
d 5
```

```
Name: CCU, dtype: int32
```

```
print(df1 + s1)
```

```
b c d
```

```
BOM  3 5 7
```

```
CCU  6 8 10
```

```
DEL  9 11 13
```

Renaming Axis Indices

- Like values in Series, axis labels can be transformed to produce new, differently labelled objects by
 - a function, or
 - mapping of some form
- The axes can also be modified in-place without creating a new data structure
- `data = DataFrame(np.arange(12).reshape((3,4)), index = ['Mumbai', 'Kolkata', 'New Delhi'], columns = ['one', 'two', 'three', 'four'])`
- Like a Series, the axis indices have a ‘map’ method
- `print(data.index.map(str.upper))` will display

```
Index(['MUMBAI', 'KOLKATA', 'NEW DELHI'], dtype='object')
```

Renaming Axis Indices

- One can assign to index, modifying the DataFrame in-place
 - `data.index = data.index.map(str.upper)`
 - `print(data)` will display
- | | one | two | three | four |
|-----------|-----|-----|-------|------|
| MUMBAI | 0 | 1 | 2 | 3 |
| KOLKATA | 4 | 5 | 6 | 7 |
| NEW DELHI | 8 | 9 | 10 | 11 |
- To create a transformed version of a data set without modifying the original, ‘rename’ method is used
 - `print(data.rename(index = str.upper, columns = str.upper))` will display

	ONE	TWO	THREE	FOUR
MUMBAI	0	1	2	3
KOLKATA	4	5	6	7
NEW DELHI	8	9	10	11

Renaming Axis Indices

- ‘rename’ can also be used, in conjunction with a dictionary-like object, to provide new values for a subset of the axis labels
- `print(data.rename(index = {'MUMBAI' : 'BENGALURU'}, columns = {'two': 'deux'}))` will display

	one	deux	three	four
BENGALURU	0	1	2	3
KOLKATA	4	5	6	7
NEW DELHI	8	9	10	11

- ‘rename’ saves having to copy the DataFrame manually and assign to its index and columns attributes.
- To modify a data set in place, one is required to pass ‘inplace = True’

- # Always returns a reference to a DataFrame
- `_ = data.rename(index = {'MUMBAI' : 'BENGALURU'}, inplace = True)`
- `print(data)` will display

	one	two	three	four
BENGALURU	0	1	2	3
KOLKATA	4	5	6	7
NEW DELHI	8	9	10	11

Summarizing

- There is a set of mathematical and statistical methods for pandas objects, most of which fall in the category of reduction or summary statistics
- These methods extract a single value (e.g. sum, mean) from a Series and a Series of values from the rows or columns of a DataFrame
- `df = DataFrame([[5.1, -0.6], [7.9, np.nan], [np.nan, 8.6], [-4.2, 2.3]], index = ['A', 'B', 'C', 'D'], columns = ['1', '2'])`
- `print(df)` will display

	1	2
A	5.1	-0.6
B	7.9	NaN
C	NaN	8.6
D	-4.2	2.3

- ‘sum’ method of DataFrame returns a Series as shown below:

• `print(df.sum())` will display column-wise sum

1 8.8
2 10.3

`print(df.sum(axis = 1))` will display row-wise sum

A 4.5
B 7.9
C 8.6
D -1.9

Summarizing

- If the entire slice (row or column) is NaN, the result is NaN, otherwise the NaN values are skipped by default
- If NaN values are not to be skipped, skipna option is to be disabled
- `print(df.mean(axis = 1, skipna = False))` will display
 - A 2.25
 - B NaN
 - C NaN
 - D -0.95
- Methods ‘idxmin’ and ‘idxmax’ return the index, where minimum or maximum values are respectively, obtained
- `print(df.idxmax())` will display `print(df.idxmin())` will display
 - 1 B
 - 2 C
 - 1 D
 - 2 A

Summarizing

- Method ‘cumsum’ returns the cumulative sum
- `print(df.cumsum())` will display

	1	2
A	5.1	-0.6
B	13.0	NaN
C	NaN	8.0
D	8.8	10.3

- Method ‘describe’ produces multiple summary statistics in one shot.
- `print(df.describe())` will display

	1	2	
count	3.000000	3.000000	
mean	2.933333	3.433333	mean = $\sigma(x) / n$
std	6.334298	4.703545	SD = $\sqrt{\sigma((x-\text{mean})^2) / (n-1)}$
min	-4.200000	-0.600000	
25%	0.450000	0.850000	
50%	5.100000	2.300000	
75%	6.500000	5.450000	
max	7.900000	8.600000	

Summarizing

- By default, the lower percentile is 25, the upper percentile is 75 and the 50 percentile is the same as the median
- If P_k is the k^{th} percentile of a set of n numbers, then it means that $k\%$ of data values are $\leq P_k$
- On non-numeric data, ‘describe’ produces alternate summary statistics
- `obj = Series(['a', 'b', 'a', 'c', 'd']*3)`
- `print(obj.describe())` will display

count	15	
unique	4	
top	a	(The most frequent element)
freq	6	(How many times the most frequent element was seen)

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 3: Data Manipulations with Pandas
Lecture 26: 10/01/2022 (15:10 hrs.)

Dr. Debranjan Sarkar

Handling Missing Data

- Missing data is very common in data analysis applications
- Pandas works with missing data very painlessly
- All of the descriptive statistics on pandas exclude missing data
- Pandas uses the floating point value NaN (Not a Number) to represent missing data in both floating point as well as non-floating point arrays
- `s1 = Series(['Kolkata', 'Mumbai', np.nan, 'Bengaluru'])`

`print(s1)` will display

0	Kolkata
1	Mumbai
2	NaN
3	Bengaluru

`print(s1.isnull())` will display

0	False
1	False
2	True
3	False

Handling Missing Data

- The built-in Python None value is treated as NaN in object arrays
- `s1 = Series(['Kolkata', 'Mumbai', np.nan, 'Bengaluru'])`
- `s1[0] = None`

`print(s1)` will display

0 None

1 Mumbai

2 NaN

3 Bengaluru

`print(s1.isnull())` will display

0 True

1 False

2 True

3 False

- Some other methods for handling missing data:
- ‘`fillna`’ fills in missing data with some value or using an interpolation method ‘`ffill`’ or ‘`bfill`’
- ‘`notnull`’ is the negation of ‘`isnull`’

Filtering out Missing Data

- `from numpy import nan as NA`

- `s1 = Series([2., NA, -5., NA, 7.5])`

- `print(s1[s1.notnull()])` or `print(s1.dropna())` will display

0 2.0

2 -5.0

4 7.5

- With DataFrame objects, rows or columns may be dropped, containing

- all missing values, or

- one or more missing values

- ‘dropna’, by default, drops the rows which contain at least one missing value

- `df = DataFrame([[2.3, 5.6, 1.9], [3.7, NA, NA], [NA, NA, NA], [NA, 4.9, -2.6]])`

`print(df)` will display

	0	1	2
0	2.3	5.6	1.9
1	3.7	NaN	NaN
2	NaN	NaN	NaN
3	NaN	4.9	-2.6

`print(df.dropna())` will display

	0	1	2
0	2.3	5.6	1.9

1	3.7	NaN	NaN
---	-----	-----	-----

2	NaN	NaN	NaN
---	-----	-----	-----

3	NaN	4.9	-2.6
---	-----	-----	------

Filtering out Missing Data

- Passing how = 'all' will drop the rows which contains all missing values
- print(df.dropna(how = 'all')) will display

	0	1	2
0	2.3	5.6	1.9
1	3.7	NaN	NaN
3	NaN	4.9	-2.6

- By passing axis = 1, columns can be dropped in a similar way
- df1 = DataFrame([[2.3, 5.6, NA], [3.7, NA, NA], [-1.2, 9.3, NA], [NA, 4.9, NA]])

print(df1) will display

	0	1	2
0	2.3	5.6	NaN
1	3.7	NaN	NaN
2	-1.2	9.3	NaN
3	NaN	4.9	NaN

print(df1.dropna(axis = 1, how = 'all')) will display

	0	1
0	2.3	5.6
1	3.7	NaN
2	-1.2	9.3
3	NaN	4.9

Filtering out Missing Data

- If it is required to keep only rows containing a certain (or above) number of observations
- This can be indicated by ‘thresh’ argument
- The number of non-missing values in a row should not be below the threshold, indicated by the argument ‘thresh’
- `df2 = DataFrame([[0.0, 1.0, NA], [2.0, NA, NA], [3.0, 5.0, 6.0], [7.0, NA, NA]])`

`print(df2)` will display

	0	1	2
0	0.0	1.0	NaN
1	2.0	NaN	NaN
2	3.0	5.0	6.0
3	7.0	NaN	NaN

`print(df2.dropna(thresh = 2))` will display

	0	1	2
0	0.0	1.0	NaN
1	2.0	3.0	5.0
2	6.0		

Filling in Missing Data

- It is preferable to fill in the missing data, rather than filtering these out
- 'fillna' method is used for this purpose

```
print(df2.fillna(0))
```

	0	1	2
0	0.0	1.0	0.0
1	2.0	0.0	0.0
2	3.0	5.0	6.0
3	7.0	0.0	0.0

```
print(df2.fillna({0: 0., 1: .5, 2: 1., 3: 1.5}))
```

	0	1	2
0	0.0	1.0	1.0
1	2.0	0.5	1.0
2	3.0	5.0	6.0
3	7.0	0.5	1.0

- 'fillna' returns a new object, but the existing object can also be modified in-place
- `_ = df2.fillna(0, inplace = True)`
- `print(df2)` will display

	0	1	2
0	0.0	1.0	0.0
1	2.0	0.0	0.0
2	3.0	5.0	6.0
3	7.0	0.0	0.0

Filling in Missing Data

- Interpolation methods (ffill, bfill) for reindexing can be used with fillna
- `df2 = DataFrame([[0.0, 1.0, NA], [2.0, NA, NA], [3.0, 5.0, 6.0], [7.0, NA, NA]])`

`print(df2)` will display

	0	1	2
0	0.0	1.0	NaN
1	2.0	NaN	NaN
2	3.0	5.0	6.0
3	7.0	NaN	NaN

`print(df2.fillna(method = 'ffill'))` will display

	0	1	2
0	0.0	1.0	NaN
1	2.0	2.0	1.0
2	3.0	3.0	5.0
3	7.0	7.0	6.0

- With fillna, lots of other things (e.g. fill with the mean value) can be done

• `s1 = Series([2., NA, -5., NA, 7.5])` # mean is $(2.0 - 5.0 + 7.5) / 3 = 1.5$

• `print(s1.fillna(s1.mean()))` will display

0	2.0
1	1.5
2	-5.0
3	1.5
4	7.5

Hierarchical Indexing

- Hierarchical indexing enables to have multiple (two or more) index levels on an axis
- It provides a way to work with higher dimensional data in a lower dimensional form
- `s1 = Series(np.random.rand(10), index = [[‘A’, ‘A’, ‘A’, ‘B’, ‘B’, ‘B’, ‘C’, ‘C’, ‘D’, ‘D’], [1, 2, 3, 1, 2, 3, 2, 3, 1, 2]])`
- `print(s1)` will display the Series with a MultiIndex as its index

A	1	0.082975
	2	0.558430
	3	0.061141
B	1	0.818464
	2	0.873858
	3	0.575057
C	2	0.772814
	3	0.717010
D	1	0.955902
	2	0.455772
- `print(s1.index)` will display the MultiIndex
`MultiIndex([(‘A’, 1),
 (‘A’, 2),
 (‘A’, 3),
 (‘B’, 1),
 (‘B’, 2),
 (‘B’, 3),
 (‘C’, 2),
 (‘C’, 3),
 (‘D’, 1),
 (‘D’, 2)]))`

Hierarchical Indexing

- With a hierarchically-indexed object, partial indexing is possible. Example:

```
print(s1['C'])
```

```
2 0.772814
```

```
3 0.717010
```

```
print(s1['B': 'C'])
```

```
B 1 0.818464
```

```
2 0.873858
```

```
3 0.575057
```

```
C 2 0.772814
```

```
3 0.717010
```

- `print(s1[:, 2])` # selection from an “inner” level

```
A 0.558430
```

```
B 0.873858
```

```
C 0.772814
```

```
D 0.455772
```

Hierarchical Indexing

- Hierarchical indexing allows rearranging of data into a DataFrame using ‘unstack’ method, the inverse operation of which is ‘stack’. Example:

```
print(s1.unstack())
```

	1	2	3
A	0.082975	0.558430	0.061141
B	0.818464	0.873858	0.575057
C	NaN	0.772814	0.717010
D	0.955902	0.455772	NaN

```
print(s1.unstack().stack())
```

A	1	0.082975
	2	0.558430
B	3	0.061141
C	1	0.818464
	2	0.873858
	3	0.575057
D	2	0.772814
	3	0.717010
E	1	0.955902
	2	0.455772

Hierarchical Indexing

- With a DataFrame, either axis can have a hierarchical index
- df= DataFrame(np.arange(12).reshape((4,3)), index = [['A', 'A', 'B', 'B'], [1, 2, 1, 2]], columns = [['CCU', 'CCU', 'PAT'], ['Green', 'Red', 'Green']])
- print(df) will display

		CCU	PAT	
		Green	Red	Green
A	1	0	1	2
	2	3	4	5
B	1	6	7	8
	2	9	10	11

- The hierarchical levels can have names, which will show up in the display
- df.index.names = ['Key1', 'Key2']; df.columns.names = ['City', 'Colour']
- print(df) will display

		City	CCU	PAT	
		Colour	Green	Red	Green
		Key1	Key2		
A	1	1	0	1	2
	2	2	3	4	5
B	1	1	6	7	8
	2	2	9	10	11

Hierarchical Indexing

- With partial column indexing, groups of columns can be selected
- print(df['CCU']) will display

	Colour	Green	Red
	Key1	Key2	
A	1	0	1
	2	3	4
B	1	6	7
	2	9	10

- A MultiIndex can be created by itself and then reused. Example:
- x = pd.MultiIndex.from_arrays([['CCU', 'CCU', 'PAT'], ['Green', 'Red', 'Green']], names = ['City', 'Colour'])
- print(x) will display

```
MultiIndex([('CCU', 'Green'),  
            ('CCU', 'Red'),  
            ('PAT', 'Green')],  
           names=['City', 'Colour'])
```

Thank you

Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: R Programming Introduction

Lecture 27: 11/01/2022

Dr. Debranjan Sarkar

R User Interface

- R is a computer language like C, C++, Python etc.
- In some languages like C, Java, FORTRAN, it is required to compile the human-readable code into machine-readable code, before the code can be run
- R is a dynamic programming language, which means R automatically interprets the code as it is run
- We use R by writing commands in the R language and asking the computer to interpret them
- To get started with R, we need to acquire our own copy of R as well as RStudio
- RStudio is a software application that helps the programmer write in R and makes R easier to use
- Both R and RStudio are free and easy to download
- Go to the site <https://cran.r-project.org/bin/windows/base/>
- Click on [Download R 4.0.2 for Windows](#) and follow the steps

RStudio

- To download RStudio, visit <https://rstudio.com/products/rstudio/download/> and follow the steps
- When you open RStudio, a window appears with three panes in it
- The largest pane is a console window
- You run your R code and see results in the console window
- In the other panes, one can have a text editor, a graphics window, a debugger, a file manager, and much more
- R Studio gives you a way to talk to your computer and R gives you a language to speak in
- So, even if you use RStudio, you still need to download R in your computer
- RStudio helps you use the version of R available in your computer, but it does not come with a version of R on its own
- When you have both R and RStudio on your computer, you can begin using R by opening the RStudio program

Objects

- Let us use R to make a virtual die
- The colon ‘:’ operator can create a group of numbers from n_s to n_e and returns the result as a vector, a one-dimensional set of numbers
- > 1:6 will display 1 2 3 4 5 6
- The vector may be saved inside an R object
- An object has a name that can be used to call up stored data
- die <- 1:6 would create an object named ‘die’ that contains the numbers 1 to 6
- To see what is stored in an object, object’s name may just be typed
- > die will display 1 2 3 4 5 6
- When an object is created, it will appear in the environment pane of RStudio
- This pane shows all the objects created since opening the RStudio
- The function ls() also shows the names of the objects already used

Objects

- Rules for naming an object:
 - A name cannot start with a number
 - A name cannot use special symbols like ^ ! \$ @ + - / *
 - Name is case sensitive ('Age' and 'age' refer to different objects)
- We have already a virtual die stored in memory. We can do a lot with this
 - die will display 1 2 3 4 5 6
 - die -1 will display 0 1 2 3 4 5
 - die /2 will display 0.5 1.0 1.5 2.0 2.5 3.0
 - die * die will display 1 4 9 16 25 36
 - R will repeat a short vector to do element-wise operations with two vectors of uneven lengths
 - die + 1:2 will display 2 4 4 6 6 8
 - die + 1:4 will display 2 4 6 8 6 8 alongwith a Warning message that longer object length is not a multiple of shorter object length

Objects

- Element-wise operations are very useful because they manipulate groups of values in an orderly way
- Traditional matrix multiplication is also possible
- Inner multiplication can be done with `%*%` operator
- die `%*%` die will display 91 ($1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6$)
- Outer multiplication can be done with `%o%` operator
- die `%o%` die will display

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
[1,]	1	2	3	4	5	6
[2,]	2	4	6	8	10	12
[3,]	3	6	9	12	15	18
[4,]	4	8	12	16	20	24
[5,]	5	10	15	20	25	30
[6,]	6	12	18	24	30	36

Objects

- `x <- 2:4;` `y <- 5:7`
- `x %o% y` will display
 - [,1] 5 [,2] 6 [,3] 7
 - [1,] 2 $2 \times 5 = 10$ $2 \times 6 = 12$ $2 \times 7 = 14$
 - [2,] 3 $3 \times 5 = 15$ $3 \times 6 = 18$ $3 \times 7 = 21$
 - [3,] 4 $4 \times 5 = 20$ $4 \times 6 = 24$ $4 \times 7 = 28$
- `y %o% x` will display
 - [,1] 2 [,2] 3 [,3] 4
 - [1,] 5 $5 \times 2 = 10$ $5 \times 3 = 15$ $5 \times 4 = 20$
 - [2,] 6 $6 \times 2 = 12$ $6 \times 3 = 18$ $6 \times 4 = 24$
 - [3,] 7 $7 \times 2 = 14$ $7 \times 3 = 21$ $7 \times 4 = 28$
- `t(x %o% y)` will display
 - [,1] [,2] [,3]
 - [1,] 10 15 20
 - [2,] 12 18 24
 - [3,] 14 21 28
- `det(x %o% y)` will display 0

Functions

- R comes with many functions that can be used to do sophisticated tasks
- Examples: round(), factorial(), mean(), sample() etc..
- `round(3.1415)` will display 3
- `factorial(4)` will display 24
- `mean(1:6)` will display 3.5
- `round(mean(die))` will display 4
- ‘sample’ function is used to simulate a roll of the die
- `sample(x = 1:4, size =2)` will return 2 (value of size) elements from the vector x (1:4)
- x and size are the names of the arguments of the function ‘sample’
- `sample(x = die, size = 1)` will return one number from the set {1,2,3,4,5,6} in a random fashion (like rolling a die)
- If size is set to 2 as in `sample(x = die, size = 2)`, it is almost as good as simulating a pair of dice

Functions

- Using names of the arguments is optional
- If the names of the arguments are not used, their sequence should be maintained e.g. `sample(die, 2)` is okay
- If the names of the arguments are used, their sequence may not be maintained e.g. `sample(size = 1, x = die)` is okay
- The names of the arguments must match with the names expected by the function e.g. `round(3.1415, precision = 2)` is wrong
- But `round(3.1415, digits = 2)` is okay and will display 3.14
- Function `args(round)` will display the arguments of the function `round`
- `args(round)` will display `function (x, digits = 0)`
- This indicates that there are two arguments of function ‘`round`’ with names `x` and `digits` and the default value for `digits` is 0

Functions

- **Sample with replacement**
- `sample(die, size = 2)` statement will **almost** simulate a pair of dice
- The word **almost** is important because, by default, `sample` builds a **sample without replacement**
- The two numbers returned by the function ‘`sample`’ are never equal which is contrary to rolling a pair of dice in real world, where each die is independent of the other
- By adding the argument `replace = TRUE`, one can have both the values same
- `sample(die, size = 2, replace = TRUE)` may result in two equal values
- If it is desirable to add up the dice, the result can be fed straight into the `sum` function as follows:
- `dice <- sample(die, size = 2, replace = TRUE)` will display 3 5 (say)
- `sum(dice)` will display 8
- If we repeat to display the value of `dice`, it will always display 3 5 because `sample` was called only once and the values were assigned to the object `dice`
- However, we may, by writing our own R function, make an object which can re-roll the dice whenever it is called

Writing our own functions

- We want to write a function ‘roll’ which may be used to roll the virtual dice and get the sum of rolling two dice
- The function basically contains the following lines of codes

```
die <- 1:6
```

```
dice <- sample(die, size =2, replace = TRUE)  
sum(dice)
```

- Each time the function roll() is called, it simulates rolling of two independent dice and returns their sum

Function constructor

- A function in R has three parts
 - A name
 - A body of code
 - A set of arguments
 - To make a new function, it is required to replicate these parts and store them in an R object
 - This can be done with the function ‘function’
 - The syntax is `new_function <- function() { }`
 - ‘function’ will build a function out of the R code placed between the braces
 - Example to build the function ‘roll’
- ```
roll <- function() {
 die <- 1:6
 dice <- sample(die, size = 2, replace = TRUE)
 sum(dice)
}
```

# Function constructor

- To use the function roll write `roll()`, which will display the sum of two independent random numbers between 1 and 6 (both inclusive)
- If the function name is typed without parentheses, R will display the code of the function
- When a function is run in R, all the code in the body of the function will be executed and the result of the last line of the code will be returned
- If the last line of the code (body) does not return a value, the function will not return any value
- So to return a value, it is to be ensured that the final line of the code returns some value
- Example:

Valid last line

`dice`

`3 + 2`

`sqrt(3)`

Invalid last line

`dice <- sample(die, size =2, replace = TRUE)`

`five <- 3 + 2`

`x <- sqrt(3)`

# Arguments

- The function ‘roll’ had no arguments and so it was called as roll()
- If we want to supply the value of die into the function ‘roll’, we may do so by an argument with name (say, bones)
- We construct the function roll2 with argument as ‘bones’ as follows:

```
roll2 <- function(bones) {
 dice <- sample(bones, size =2, replace = TRUE)
 sum(dice)
}
```

- Function roll2 will work if the value of bones is supplied, when calling the function
- roll2(bones = 1:4) will display the sum of two independent random numbers between 1 and 4 (both inclusive)

# Arguments

- R allows a default value of the argument as follows:

```
roll2 <- function(bones = 1:6) {
 dice <- sample(bones, size = 2, replace = TRUE)
 sum(dice)
}
```

- If the function is called without argument as roll2(), it will display the sum of two independent random numbers between 1 and 6 (both inclusive)
- A function may have more than one arguments as shown below:

```
fun_name <- function(arg1, arg2, ..., argn) {
 Body of the function
}
```

- It is to be called as fun\_name(arg1, arg2, ..., argn)
- The following parts of an R function are important to note
  - The name
  - The body
  - The arguments
  - The default values for the arguments
  - Last line of the code

# Scripts

- One can create a draft of one's code, using an R script
- One can edit a part of code very easily, if the draft of the code is available to start with
- An R script is just a plain text file that is saved in R code
- The R script can be opened in RStudio by going to File > New file > R script in the menu bar
- RStudio will open a fresh script above the console pane
- Scripts are handy for editing and proofreading the code
- To save a script, click the scripts pane, and then go to File > Save As in the menu bar
- RStudio has many built-in features to facilitate work with scripts
  - One can automatically execute a line of code in a script by clicking the Run button (or Ctrl Return)
  - One can run the entire script by clicking the Source button
- Scripts facilitates painless writing of multi-line code contrary to the console's single-line command line
- RStudio has a tool to help build functions
  - Highlight the lines of code in the R script that is to be turned into a function
  - Then click Code > Extract Function in the Menu Bar
  - RStudio will ask for a function name to use and then wrap the code in a function call

# R Data Structure

# Atomic Vector

- The most simple type of object in R is an atomic vector
- It is just a simple vector of data, but it is called atomic, as most structures in R are built from atomic vectors
- A function ‘c’ (c stands for concatenate, or collect or combine) can create an atomic vector by grouping some values of data together

```
die <- c(1, 2, 3, 4, 5, 6)
```

- The function `is.vector()` tests whether an object is an atomic vector

|                             |                   |
|-----------------------------|-------------------|
| <code>is.vector(die)</code> | will display TRUE |
|-----------------------------|-------------------|

- An atomic vector can also be made with just one value (atomic vector of length 1)

```
nine <- 9
```

|                              |                   |
|------------------------------|-------------------|
| <code>is.vector(nine)</code> | will display TRUE |
|------------------------------|-------------------|

|                           |                |
|---------------------------|----------------|
| <code>length(nine)</code> | will display 1 |
|---------------------------|----------------|

|                          |                |
|--------------------------|----------------|
| <code>length(die)</code> | will display 6 |
|--------------------------|----------------|

- Each atomic vector stores its values as one-dimensional vector
- Each atomic vector can only store one type of data
- Different types of data can be saved in R by using different types of atomic vectors

# Atomic Vector: Double

- There are six basic types of atomic vectors:
  - double, integer, character, logical, complex, and raw
- A double vector stores regular numbers (+ve or -ve) viz +1, -3.14, -10, 1.15, etc.
- `die <- c(1, 2, 3, 4, 5, 6)`
- The function `typeof()` returns the type of object in the argument of the function
- `typeof(die)` will display “double”
- Some R functions refer to double as numeric
- ‘Double’ is a computer science term, which refers to the specific number of bytes used to store a number
- The term ‘numeric’ is more intuitive when doing data science

# Atomic Vector: Integer

- Integer vectors store integers
- Integers are, by default, stored as a double object, as already shown
- However, an integer vector can be specifically created by typing an integer number followed by L
- `x <- c(1L, 2L, 6L)`
- `typeof(x)` will display “integer”
- 64 bits of memory is allocated to store each ‘double’ value
- This allows a lot of precision – each double is precise to about 16 digits
- This introduces a little bit of rounding error, which will go unnoticed
- In the following example, the result of the expression is expected to be zero, but
- `sqrt(3) ^ 2 - 3` will display `-4.440892e-16`, which is very close to 0 but not 0
- This error is known as floating-point error, which occurs due to floating-point arithmetic
- Floating-point arithmetic is not a feature of R, it is a feature of computer programming

# Atomic Vector: Character

- A character vector stores small pieces of text
- A character vector may be created by typing a character or string of characters surrounded by quotes (single or double)
- `t <- c('x', 'y')`
- `t` will display “x”, “y”
- `typeof(t)` will display “character”
- The individual elements of a character vector are strings

# Atomic Vector: Logical

- Logical vectors store TRUEs and FALSEs
- $9 > 16$  will display FALSE
- `logi <- c(FALSE, TRUE, FALSE)`
- `logi` will display FALSE, TRUE, FALSE
- `typeof(logi)` will display “logical”
- R also understands that T and F are respectively the shorthand notations for TRUE and FALSE
- `typeof(T)` will display “logical”

# Atomic Vector: Complex and Raw

- **Complex vectors** store complex numbers
  - To create a complex vector, add an imaginary term to a number with i
  - `comp <- c(1 + 1i, -2 +3i, 5 + 0i)`
  - `typeof(comp)` will display “complex”
- 
- **Raw vectors** store raw bytes of data
  - One can make an empty raw vector of length n with `raw(n)`
  - `charToRaw(char)` converts the ASCII code of the character char to raw data
  - `raw(3)` will display 00 00 00
  - `typeof(raw(3))` will display “raw”
  - `xx <- raw(2)`
  - `xx[1] <- charToRaw("A")`
  - `xx[2] <- charToRaw("5")`
  - `xx` will display 41 35
- Note that the ASCII code for the characters “A” and “5” are  $41_{16}$  and  $35_{16}$  respectively

# Attributes

- An attribute is a piece of information that can be attached to an atomic vector (or any R object)
- The most common attributes are names, dimensions (dim), and classes
- The attribute would not affect any of the values of the object and it will not appear when the object is displayed
- An attribute may be thought of as “metadata” (data about data)
- R will normally ignore this metadata but some R functions will check for specific attributes
- These functions may use the attributes to do special things with the data
- `Function attributes(obj) will return the attributes of an object obj`
- The atomic vector ‘die’ has no attributes as it has not been given any
- `attributes(die) will display NULL`
- In R, NULL represents the null set or an empty object
- `NULL is returned by functions whose values are undefined`
- A NULL object may be created by typing NULL in capital letters

# Attributes: Names

```
names(die) # names attributes (now NULL) of die can be looked up by the 'names' function
names(die) <- c("one", "two", "three", "four", "five", "six") # Names can be given to die
names(die) will display "one", "two", "three", "four", "five", "six"
attributes(die) will now display $names
 [1] "one" "two" "three" "four" "five" "six"
die one two three four five six # will display the names above the elements of 'die'
 1 2 3 4 5 6
die - 1 one two three four five six # The names are not affected
 0 1 2 3 4 5
names(die) <- c("un", "deux", "trois", "quatre", "cinq", "six") # names attribute may be changed
die un deux trois quatre cinq six
 1 2 3 4 5 6
names(die) <- NULL # will remove the names attribute, set it to NULL
die 1 2 3 4 5 6
```

# Attributes: Dim

- Atomic vector may be transformed into an n-dimensional array by giving it a dimensions attribute with 'dim'

- `dim(die) <- c(2, 3)` # The object 'die' is reorganized into a 2 X 3 matrix

- die will display

|      |      |      |      |
|------|------|------|------|
|      | [,1] | [,2] | [,3] |
| [1,] | 1    | 3    | 5    |
| [2,] | 2    | 4    | 6    |

- `dim(die) <- c(1, 2, 3)` # 1 X2 X 3 hypercube is created, which will be displayed slice by slice

- die will display

|       |      |      |
|-------|------|------|
| , , 1 | [,1] | [,2] |
| [1,]  | 1    | 2    |
| , , 2 | [,1] | [,2] |
| [1,]  | 3    | 4    |
| , , 3 | [,1] | [,2] |
| [1,]  | 5    | 6    |

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: R Data Structure  
Lecture 28: 17/01/2022

Dr. Debranjan Sarkar

# Matrices

- The first value in dim represents rows and the second value represents columns
- R always fills up each matrix by columns, and not rows.
- To have more control over this process, one can use function ‘matrix’ or ‘array’
- Matrices store values in a 2-dimensional array
- `m <- matrix(die, ncol = 3)`                          or                          `m <- matrix(die, nrow = 2)`
- m will display                          [,1]                  [,2]                  [,3]  
                                        [1,]                  1                  3                  5  
                                        [2,]                  2                  4                  6
- By default, ‘matrix’ fills up the matrix column by column, which may be changed using argument `byrow = TRUE`
- `m <- matrix(die, ncol = 3, byrow = TRUE)`
- m will display                          [,1]                  [,2]                  [,3]  
                                        [1,]                  1                  2                  3  
                                        [2,]                  4                  5                  6

# Arrays

- The ‘array’ function creates an n-dimensional array
- The first argument is an atomic vector and the second argument is a vector of dimensions, called dim
- ar <- array(c(51:58, 61:68, 71:78), dim= c(2, 4, 3))
- ar will display

|  |  |  |       |                     |
|--|--|--|-------|---------------------|
|  |  |  | , , 1 |                     |
|  |  |  |       | [,1] [,2] [,3] [,4] |
|  |  |  | [1,]  | 51 53 55 57         |
|  |  |  | [2,]  | 52 54 56 58         |
|  |  |  | , , 2 |                     |
|  |  |  |       | [,1] [,2] [,3] [,4] |
|  |  |  | [1,]  | 61 63 65 67         |
|  |  |  | [2,]  | 62 64 66 68         |
|  |  |  | , , 3 |                     |
|  |  |  |       | [,1] [,2] [,3] [,4] |
|  |  |  | [1,]  | 71 73 75 77         |
|  |  |  | [2,]  | 72 74 76 78         |

# Attributes: Classes

- A class is a special case of an atomic vector
- Example: The die matrix (or array) is a special case of a double vector
- Every element of the matrix (or array) is a double, but the elements are arranged into a new structure
- When the dimensions of a die have been changed, a ‘class’ attribute is added
- `dim(die) <- c(1, 2, 3)`
- `typeof(die)` will return “double” and `class(die)` will return “array”
- An object’s class attribute will not always appear when the ‘attributes’ function is called
- `attributes(die)` will show
  - \$dim
  - [1] 1 2 3
- When ‘class’ is applied to objects not having a class attribute, it will return a value, based on the object’s atomic type
- `class("Hello")` will return “character”
- `class(4)` will return “numeric”

# Dates and Times

- R uses a special class to represent dates and times
- now <- Sys.time() displays the time on the computer as follows  
"2020-10-04 11:07:04 IST"
- Though the time looks like a character string, but the data type of 'now' is actually "double" and its class is "POSIXct" "POSIXt" (it has two classes)
- In POSIXct framework, time is represented by the number (double vector) of seconds elapsed after 00:00 hrs. on 01/01/1970
- This double vector can be seen by unclass(now) as 1601789824
- This double vector is given a class attribute containing two classes - "POSIXct" and "POSIXt"
- One can give the POSIXct class to random R objects
- To know the time  $10^6$  seconds after 00:00 hrs. on 01/01/1970, we do the following:
- deltaT <- 1000000
- class(deltaT) <- c("POSIXct", "POSIXt")
- deltaT will display "1970-01-12 19:16:40 IST"

# Factors

- Factor is one class of data which is very much ubiquitous in R
- Gender (values male or female) is a factor, based on which some peculiar ordering is to be made (ladies first!!)
- To make a factor, an atomic vector is passed into the ‘factor’ function
- gender <- factor(c('male', 'female', 'female', 'male'))
- R will recode the data in the vector as integers and store the results in an integer vector  
    `typeof(gender)`               =>       “integer”
- R will also add a ‘levels’ attribute to the integer, which contains a set of labels for displaying the factor values, and a ‘class’ attribute, which contains the class ‘factor’
- `attributes(gender)` will display
  - \$levels
    - “female” “male”
  - \$class
    - “factor”

# Factors

- How R stores the factor can be seen with ‘unclass’
- `unclass(gender)` will display [1] 2 1 1 2  
attr(“levels”)  
[1] “female” “male”
- When the factor is displayed, R uses the levels attribute.
- It will display each 1 as female and 2 as male, the second label  
`gender` will display male female female male  
Levels: female male
- Factors make it easy to put the variables into a statistical model as the variables are already coded (as numbers)
- Factors can be confusing as they look like strings but behave like integers
- A factor may be converted to a character string by `as.character(gender)`
- This displays “male” “female” “female” “male”

# Coercion

- If it is tried to put multiple types of data into a vector, R will convert the elements to a single type of data
- Rules for coercion of data types in R
- If a character string is present in an atomic vector, R will convert everything else in the vector to character string
- If a vector contains only logicals and numbers, R will convert the logicals to numbers (TRUE becomes 1 and FALSE becomes 0)
- This arrangement preserves information and one can spot the origins of “TRUE” and “5”
- Similarly, one can back-transform a vector of 1s and 0s to TRUEs and FALSEs
- Same coercion rule is applied when one tries to do math with logical values
- `sum(c(TRUE, TRUE, FALSE, FALSE)) > sum(c(1, 1, 0, 0)) > 2`
- R can convert data from one type to another with the ‘as’ function
- `as.character(3) => "3"`      `as.logical(1) => TRUE`      `as.numeric(FALSE) => 0`

# Lists

- Lists are like atomic vectors because they group data into a one-dimensional set
- Lists group together R objects (e.g. atomic vectors and other lists) but not the individual values
- Like atomic vectors, lists are used as building blocks to create many more sophisticated objects
- We show below a list of three elements:
  - Numeric vector of length 30 as the first element
  - A character vector of length 1 in the second element
  - Another list of length 2 as the third element
- To create a list, we use the function `list` as `lst <- list(50:79, "H", list(TRUE, FALSE))`
- `lst` will display `[[1]]`

```
[1] 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
[24] 73 74 75 76 77 78 79
[[2]]
[1] "H"
[[3]]
[[3]][[1]]
[1] TRUE
[[3]][[2]]
[1] FALSE
```

# Data Frames

- Data Frame is a special class (two-dimensional version) of a list
- This is the most useful storage structure for data analysis
- A data frame may be thought of as R's equivalent to the Excel spreadsheet
- Data frames group vectors together into a two-dimensional table
- Each vector becomes a column in the table
- Thus each column of a data frame may contain different types of data
- But within a column, every cell must be of the same type of data

|   |        |      |       |
|---|--------|------|-------|
| 1 | “HITK” | 4679 | FALSE |
| 2 | “JU”   | 6190 | TRUE  |
| 3 | “NIT”  | 5916 | TRUE  |
| 4 | “IEM”  | 3881 | FALSE |

# Data Frames

- To create a data frame, function `data.frame` is used as follows:
- `df <- data.frame(Institute = c("HITK", "JU", "NIT", "IEM"),  
Number = c(4679, 6190, 5916, 3881),  
Govt = c(FALSE, TRUE, TRUE, FALSE))`

- `df` will show

|   | Institute | Number | Govt  |
|---|-----------|--------|-------|
| 1 | HITK      | 4679   | FALSE |
| 2 | JU        | 6190   | TRUE  |
| 3 | NIT       | 5916   | TRUE  |
| 4 | IEM       | 3881   | FALSE |

# Data Frames

- Each vector should be given a name as Institute, Number and Govt
- Each vector must be of same length
- A data frame is of type ‘list’ with class data.frame
- With the ‘str’ function, one can see what types of objects are grouped together by a list (or data frame)
- `typeof(df) => "list"`                              `class(df) => "data.frame"`
- `str(df)` will display  

```
'data.frame': 4 obs. of 3 variables:
 $ Institute : chr "HITK" "JU" "NIT" "IEM"
 $ Number : num 4679 6190 5916 3881
 $ Govt : logi FALSE TRUE TRUE FALSE'
```
- To create a large data frame, one is required to type a lot
- To avoid lot of typing of data, one can load the data from a comma-separated values file or CSV file
- CSV files are plain-text files that can be opened by a text editor
- RStudio facilitates users to load data from a plain-text file and save into it

# References

- “Hands-on programming with R”, Garrett Grokemund, O'Reilly
- “Advanced R”, Hadley Wickham, CRC Press

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: Computing with R  
Lecture 29: 18/01/2022

Dr. Debranjan Sarkar

# Computing with R

# Selection

- To extract a value or a set of values from a data frame, write the data frame's name, followed by a pair of third brackets `df[I1, I2]`
- I<sub>1</sub> and I<sub>2</sub> are the indexes to subset the rows and columns respectively
- One can create indexes with +ve integers, -ve integers, zero, blank space, logical values, names
- **Positive integers as index:**

- `df[1,1]` will display

“HITK”

- `df[2,c(1, 2, 3)]` will display

|   | Institute | Number | Govt |
|---|-----------|--------|------|
| 2 | JU        | 6190   | TRUE |

- `df[c(1,1), c(1,2,3)]` will display

|     | Institute | Number | Govt  |
|-----|-----------|--------|-------|
| 1   | HITK      | 4679   | FALSE |
| 1.1 | HITK      | 4679   | FALSE |

- `vec <- c(6, -1, 3, 8, 2, -7)`

# Indexing for vector

- `vec[2:4]` will display

-1 3 8

# Selection

- Note that unlike Python, indexing begins with 1
- If two or more columns are selected from a data frame, we get a new data frame
- `df[2:3, 1:2]` will display

|   | Institute | Number |
|---|-----------|--------|
| 2 | JU        | 6190   |
| 3 | NIT       | 5916   |

- However, if one column is selected, R returns a vector
- `df([1:2, 2])` will display 4679 6190
- If it is preferred to have a data frame instead of a vector, optional argument `drop = FALSE` is to be added
- `df([1:2, 2], drop = FALSE)` will display

|   | Number |
|---|--------|
| 1 | 4679   |
| 2 | 6190   |

# Selection

- **Negative integers as index:**

- R will return every element except the elements in the negative index
- df[ $-(2:3)$ , 1:3] will display

|   | Institute | Number | Govt  |
|---|-----------|--------|-------|
| 1 | HITK      | 4679   | FALSE |
| 4 | IEM       | 3881   | FALSE |

- Pairing a negative integer with a positive integer in the same index is not allowed
- df[c( $-2,3$ ), 1] will display Error “only 0's may be mixed with negative subscripts”
- However, a negative integer and a positive integer may be used in different indexes (row and column)
- df[-2, 1] will display "HITK" "NIT" "IEM"

# Selection

- **Zero as index:**
  - R will return nothing from a dimension when zero is used as an index
  - This creates an empty object
  - `df[0, 0]` will display “data frame with 0 columns and 0 rows”
  - `df[2, 0]` will display “data frame with 0 columns and 1 row”
  - `df[0, 1]` will display `character(0)`
  - `df[0, 2]` will display `numeric(0)`
  - `df[0, 3]` will display `logical(0)`
- **Blank spaces as index:**
  - Blank space is used to extract every value in a dimension
  - This is useful for extracting entire rows or columns from a data frame
  - `df[1, ]` will display

|   | Institute | Number | Govt  |
|---|-----------|--------|-------|
| 1 | HITK      | 4679   | FALSE |

# Selection

- **Logical values as index:**
- If a vector of TRUEs and FALSEs is supplied as a row (or column) index, R will return each row (or column) that corresponds a TRUE
- `df[1, c(TRUE, FALSE, TRUE)]` will display

|   | Institute | Govt  |
|---|-----------|-------|
| 1 | HITK      | FALSE |

- `rows <- c(TRUE, FALSE, FALSE, TRUE)`
- `df[rows, ]` will display

|   | Institute | Number | Govt  |
|---|-----------|--------|-------|
| 1 | HITK      | 4679   | FALSE |
| 4 | IEM       | 3881   | FALSE |

- `vec <- c(6, -1, 3, 8, 2, -7)` # creating a vector vec
- `vec[c(T, F, F, T, T, F)]` will display 6 8 2

## Names as index:

- `df[1, c("Institute", "Number")]` will display
- `df[, c("Number")]` will display 4679 6190 5916 3881

|   | Institute | Number |
|---|-----------|--------|
| 1 | HITK      | 4679   |

# Example: Playing cards

- Let us define a data frame for playing cards
- deck <- data.frame(

```
face = c("king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five", "four", "three", "two",
"ace", "king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace",
"king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace",
"king", "queen", "jack", "ten", "nine", "eight", "seven", "six", "five", "four", "three", "two", "ace"),

suit = c("spades", "spades", "spades", "spades", "spades", "spades", "spades", "spades",
"spades", "spades", "spades", "spades", "clubs", "clubs", "clubs", "clubs", "clubs", "clubs",
"clubs", "clubs", "clubs", "clubs", "clubs", "clubs", "diamonds", "diamonds", "diamonds", "diamonds",
"diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds", "diamonds",
"diamonds", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts", "hearts",
"hearts", "hearts", "hearts", "hearts"),

value = c(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9,
8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1))
```

- deck will display

|   | face  | suit   | value |               |
|---|-------|--------|-------|---------------|
| 1 | king  | spades | 13    |               |
| 2 | queen | spades | 12    |               |
| 3 | jack  | spades | 11    |               |
| 4 | ten   | spades | 10    | and so on.... |

# Example: Playing cards

- `head(deck)` will display first six rows

|   | face  | suit   | value |
|---|-------|--------|-------|
| 1 | king  | spades | 13    |
| 2 | queen | spades | 12    |
| 3 | jack  | spades | 11    |
| 4 | ten   | spades | 10    |
| 5 | nine  | spades | 9     |
| 6 | eight | spades | 8     |

- `tail(deck)` will display last six rows

|    | face  | suit   | value |
|----|-------|--------|-------|
| 47 | six   | hearts | 6     |
| 48 | five  | hearts | 5     |
| 49 | four  | hearts | 4     |
| 50 | three | hearts | 3     |
| 51 | two   | hearts | 2     |
| 52 | ace   | hearts | 1     |

- `head(deck, 3)` will display first three rows

# Example: Playing cards

- We write a function ‘deal’ with argument ‘cards’ which returns the first row of the data frame (i.e. the card at the top of the card deck)

```
deal <- function(cards){
 cards[1,]
}
```

- `deal(deck)` will display

|   | face | suit   | value |
|---|------|--------|-------|
| 1 | king | spades | 13    |

- To shuffle a deck of cards, we randomly rearrange the order of the cards
- We create 52 numbers (from 1:52, without replacement) in random order
- `random <- sample(1:52, size = 52)` will display

```
[1] 10 14 5 46 23 40 43 1 39 35 49 52 30 8 47 26 21 32 29 42 2 13 22
[24] 51 27 9 38 3 6 41 11 19 4 28 48 20 24 45 15 12 36 16 33 50 7 34
[47] 44 25 17 18 31 37
```

- `shuffled_deck <- deck[random, ]`; `head(shuffled_deck, 4)` will display

|    | face  | suit   | value |
|----|-------|--------|-------|
| 10 | four  | spades | 4     |
| 14 | king  | clubs  | 13    |
| 5  | nine  | spades | 9     |
| 46 | seven | hearts | 7     |

# Example: Playing cards

- Write a ‘shuffle’ function to take a data frame and return a shuffled copy of the data frame

```
shuffle <- function(cards){
 random <- sample(1:52, size = 52)
 cards[random,]
}
```

- `deal(deck)` will display

|   | face | suit   | value |
|---|------|--------|-------|
| 1 | king | spades | 13    |

- `deck2 <- shuffle(deck)`

- `deal(deck2)` will display

|    | face | suit     | value |
|----|------|----------|-------|
| 39 | ace  | diamonds | 1     |

# Dollar (\$) sign and Double Brackets

- Values from data frames and list can be extracted with the \$ syntax
- To select a column from a data frame (df), df\$Institute may be used
- df\$Institute will display "HITK" "JU" "NIT" "IEM"
- df\$Number will display 4679 6190 5916 3881
- mean(df\$Number) will display 5166.5
- median(df\$Number) will display 5297.5
- \$ notation may be used with the elements of a list, if they have names
- `Ist <- list( numb = c(5,6), logi = FALSE, str = c("x", "y", "z") )`
- Ist will display \$numb  
[1] 5 6  
\$logi  
[1] FALSE  
\$str  
[1] "x" "y" "z"

# Dollar (\$) sign and Double Brackets

- `lst[1]` will result in a smaller list with one element and show `$numb`  
`[1] 5 6`
- Many R functions do not work with lists e.g `sum(lst[1])` will give rise to error
- When \$ notation is used, R returns the selected values only with no list structure
- `lst$numb` will return 5 6 and `sum(lst$numb)` will return 11
- If the elements of the list do not have names or names are not to be used, we use double brackets `[]` which acts similar to \$ notation
- `lst[[1]]` will return 5 6 and `sum(lst[[1]])` will return 11
- Subsetting a list with single bracket `[]` notation returns a smaller list
- Subsetting a list with double bracket `[]` notation returns just the values that were inside an element of the list
- `lst[["numb"]]` will display 5 6 but `lst["numb"]` will display `$numb`  
`5 6`

# Modification

- Changing values in place:

- `vec <- c(0, 0, 0, 0, 0, 0)` # vec is an atomic vector
- `vec[2] <- 100` # modify the 2<sup>nd</sup> value only to 100
- `vec[c(2, 4, 6)] <- c(1, 3, 5)` # modify the 2<sup>nd</sup>, 4<sup>th</sup> and 6<sup>th</sup> elements
- `vec` will display 0 1 0 3 0 5
- `vec[3:5] <- vec[3:5] + 2` # vec now becomes 0 1 2 5 2 5
- `vec[7] <- 0` # the object vec will be expanded to accommodate the new value
- New columns may be added to a data frame:

```
df$PIN <- c(700107, 700032, 713209, 700091)
```

- df will now display
- |   |      | Institute | Number | Govt   | PIN |
|---|------|-----------|--------|--------|-----|
| 1 | HITK | 4679      | FALSE  | 700107 |     |
| 2 | JU   | 6190      | TRUE   | 700032 |     |
| 3 | NIT  | 5916      | TRUE   | 713209 |     |
| 4 | IEM  | 3881      | FALSE  | 700091 |     |

# Modification

- The statement `df$Govt <- NULL` will remove the column Govt

- df will now display

|   | Institute | Number | PIN    |
|---|-----------|--------|--------|
| 1 | HITK      | 4679   | 700107 |
| 2 | JU        | 6190   | 700032 |
| 3 | NIT       | 5916   | 713209 |
| 4 | IEM       | 3881   | 700091 |

- `df$Number[c(1,3)] <- c(5000, 5000)`

or `df$Number[c(1,3)] <- 5000`

- Now df will display

|   | Institute | Number | PIN    |
|---|-----------|--------|--------|
| 1 | HITK      | 5000   | 700107 |
| 2 | JU        | 6190   | 700032 |
| 3 | NIT       | 5000   | 713209 |
| 4 | IEM       | 3881   | 700091 |

- This is in-place modification

- The same technique is applicable for vector, matrix, array, list or data frame

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: Computing with R  
Lecture 30: 19/01/2022

Dr. Debranjan Sarkar

# Logical Subsetting

- `vec <- c(1, 0, -3, 2, 9, 2)`
- `vec[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]` will display 1 -3 2 2
- One can create a logical vector by logical tests, performed by logical operators viz.  
`>, >=, <, <=, ==, !=, %in%`
- `vec > 0` will display                   TRUE FALSE FALSE TRUE TRUE TRUE
- `vec == 0` will display                   FALSE TRUE FALSE FALSE FALSE FALSE
- Note that = is an assignment operator (like `<-`) and == is equality operator
- `%in%` tests whether the value(s) on the LHS are in the vector in the RHS
- `c(1, 2, 3, 4) %in% c(3, 4, 5)` will display FALSE FALSE TRUE TRUE
- Any two R objects can be compared with a logical operator
- Normally two objects of the same type are compared
- If two objects of different data types are compared, the objects are coerced to the same type before actual comparison is made

# Logical Subsetting

- To modify the values of all aces (from 1 to 14) in a shuffled deck (deck2), we do the following:
  - `deck2$face == "ace"` will return a logical vector of 52 elements, in which only 4 elements are TRUE (because there are 4 aces in random position)
  - `sum(deck2$face == "ace")` will display 4
  - `deck2$value[deck2$face == "ace"] <- 14` will modify the values of aces
  - So we need not know where in the data set a value exists
  - In another game, every card has a value zero, except that each card in the suit of "hearts" has a value 1 and the queen of spades is worth 12
  - The first part of the problem can be solved by the following statements:
    - `deck3 <- deck`
    - `deck3$value <- 0`
    - `deck3$value[deck3$suit == "hearts"] <- 1`
  - To solve the second part of the problem, we use Boolean operators

# Boolean Operators

- The results of multiple logical tests can be collapsed into a single TRUE or FALSE
- R has six Boolean operators:

|       |                                |                                         |
|-------|--------------------------------|-----------------------------------------|
| • &   | cond1 & cond2                  | Both cond1 and cond2 true?              |
| •     | cond1   cond2                  | Is one or more of cond1 and cond2 true? |
| • xor | xor(cond1, cond2)              | Is exactly one of cond1 and cond2 true? |
| • !   | !cond1                         | Is cond1 false?                         |
| • any | any(cond1, cond2, cond3, ....) | Are any of the conditions true?         |
| • all | all(cond1, cond2, cond3, ....) | Are all of the conditions true?         |
- When used with vectors, Boolean operators follow the element-wise execution
- `x <- c(1, 2, 3);`      `y <- c(1, 2, 3);`      `z <- c(1, 2, 4)`
- `x == y` will display      TRUE TRUE TRUE
- `y == z` will display      TRUE TRUE FALSE
- `x == y & y == z` will display      TRUE TRUE FALSE
- The following two statements will solve the second part of the previous problem
- `queenOfSpades <- deck3$face == "queen" & deck3$suit == "spades"`
- `deck3$value[queenOfSpades] <- 12`

# Handling missing information

- Information is missing when the value is not known because the measurement was lost, corrupted or never taken place
- The NA character is used for missing information
- To handle missing information, most functions in R have an optional argument na.rm, which stands for NA remove
- If na.rm = TRUE, the NAs in the data set are ignored
- `mean(c(NA, 1:50), na.rm = TRUE)` will display 25.5
- The function ‘is.na’ tests whether a value is an NA
- `is.na(NA)` will display TRUE
- `vec <- c(1, 2, NA, 3, 4 ,NA, 5)`
- `is.na(vec)` will display FALSE FALSE TRUE FALSE FALSE TRUE FALSE

# Conditionals

- The syntax of 'if' statement, to conditionally execute code, is as follows:

```
if (condition) {
 # code to be executed when condition is TRUE
} else {
 # code to be executed when condition is FALSE
}
```

# Conditionals

- Write a function which returns a logical vector describing whether each element of a vector is named:

```
has_name <- function(x) {
 nms <- names(x) # function 'names' returns the names of the vector x
 if (is.null(nms)) {
 rep(FALSE, length(x)) # rep(FALSE, n) replicate FALSE n times
 } else {
 !is.na(nms) & nms != ""
 }
}
```

- `vec<- c(av = c(1,2), "TRUE", numb = 5)`
- `has_name(vec)` will display TRUE TRUE FALSE TRUE

# Conditionals

- The condition must evaluate to either TRUE or FALSE
- If it is a vector, a warning message is issued
- If it is an NA, an error message is issued
- || (or) and && (and) may be used to combine multiple logical expressions
- | or & should never be used in an if statement
- Multiple Conditions:

```
if (condition1) {
 # do this
}
else if (condition2) {
 # do something else
}
else {
 # do this if none of the above conditions are satisfied
}
```

# Conditionals

- `switch()` function eliminates the use of a long series of chained if statements and allows to evaluate the selected code based on position or name:

```
arith <- function(x, y, op) {
 switch(op,
 plus = x + y,
 minus = x - y,
 times = x * y,
 divide = x / y,
 message("Unknown op!")
)
}
```

- `arith(4, 5, "plus")` will display 9
- `arith(4, 5, "times")` will display 20

# Conditionals: Code Style

- An opening curly brace { should always be followed by a new line and never go on its own line
- A closing curly brace } should always go on its own line, unless it is followed by else
- Always indent the code inside curly braces

*# Good*

```
if (y < 0 && debug) {
 message("Y is negative")
}

if (y == 0) {
 log(x)
} else {
 y ^ x
}
```

*# Bad*

```
if (y < 0 && debug)
 message("Y is negative")

if (y == 0) {
 log(x)
}
else {
 y ^ x
}
```

# Conditionals: Code Style

- The curly braces may be dropped, if you have a very short if statement that can fit on one line:

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

- However, the full form is easier to read

```
if (y < 20) {
 x <- "Too low"
} else {
 x <- "Too high"
}
```

# Environments

- It is important to know how R stores, looks up, and manipulates objects
- R does all these things with the help of an environment system
- Environment system in R is hierarchical in nature and similar to file system in computer
- A computer stores (or saves) files in a folder, which is, in turn, saved in another folder and so on. This forms a hierarchical file system
- When the computer needs to open up a file, it must first look up this file system
- R uses a similar system to save R objects
- Each object is saved inside an environment (like folder in file system)
- An environment is a list-like object and it resembles a folder in a computer
- Each environment is connected to a parent environment, which is connected to its parent environment and so on
- R's environments exist in the RAM and not in the file system
- R's environments are not technically saved inside one another
- The connection between an environment and its parent is one-way – there is no way to look at one environment and tell what its “children” are
- One cannot search down R's environment tree
- Otherwise, R's environment system works similar to a file system

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: Computing with R  
Lecture 31: 24/01/2022

Dr. Debranjan Sarkar

# The Active Environment

- At any point of time, R works closely with a single environment
- If an object is created, the new object is stored in this environment
- If an object is to be found, this environment will first be tried to look for the object
- This environment is called active environment
- The active environment is usually the global environment, but this may change when a function is run
- The current active environment may be seen by the ‘environment’ function
- `environment()` will display      `<environment: R_GlobalEnv>`
- Global environment is the active environment for every command that is run in the command line
- Any object, created in the command line, is saved in the global environment
- The global environment may be thought of as the user workspace
- When an object is called in the command line, R will look for the object, first in the global environment

# Scoping Rules

- When an object is called in the command line, R will look for the object, first in the global environment
- But if the object is not found in the global environment, R follows a series of rules (known as Scoping Rules) to look up the object
- Scoping Rules:
  - [1] R looks for the objects in the current active environment
  - [2] When the user works at the command line, the active environment is the global environment. So R looks up the objects, called at the command line, in the global environment
  - [3] When R does not find an object in an environment, R looks in the environment's parent environment, then the parent of the parent, and so on, until R finds that object or reaches the empty environment
- Note that the functions are a type of object in R
- So, R will store and look up functions, the same way it stores and looks up other objects, by searching for them by name in the environment tree

# Assignment

- When a value is assigned to an object, R saves the value in the active environment under the object's name
- If an object with the same name already exists in the active environment, R will overwrite it
- For example, let there be already an object named 'x' in the global environment whose value is "hello"
- x will display "hello"
- If an object named 'x' is saved in the global environment, R will overwrite on the already existing object in the global environment
- x <- "How are you?"
- Now, x will display "How are you?"
- This poses a problem when R runs a function
- Functions (e.g. roll) may save temporary objects (viz. die and dice)
- If R saves these objects in the active environment, it might overwrite existing objects
- In order to avoid that, whenever a function is run, R creates a new active environment wherein the temporary objects are stored

```
roll <- function() {
 die <- 1:6
 dice <- sample(die, size =2, replace = TRUE)
 sum(dice)
}
```

# Evaluation of a function

- The new environments, created by R during each execution of a function, are called runtime environments
- The environment, in which the function was first created, is called origin environment
- After execution of a function, R will return to the origin environment alongwith the result of the function
- All of the function's runtime environments will use the origin environment as a parent
- If a function is created at the command line, the origin environment is the global environment
- If a function is created in some other environment, the origin environment is that environment
- If a function creates some objects within it, those objects are stored in the runtime environment as this is the active environment when these objects are created
- This is how R ensures that a function does not overwrite anything that it should not
- If a function has arguments, R will copy over each argument to the runtime environment
- The argument will appear as an object that has the name of the argument and the value that has been provided at the time of calling the function
- This ensures that a function will be able to find and use each of its arguments

# Evaluation of a function

- Before a function is called, R is working in an active environment (calling environment)
- When the function is called from this calling environment, R responds by setting up a new runtime environment
- This environment is a child of the function's origin environment
- R copies each of the function's arguments into the runtime environment and then makes the runtime environment the new active environment
- Next, the code in the body of the function is run
- If the code creates any objects, these are stored in active (i.e. runtime) environment
- If the code calls any objects, R uses its scoping rules to look them up
- R will search the runtime environment, then the parent of the runtime environment (i.e. the origin environment), then the parent of the origin environment, and so on.
- Notice that the calling environment might not be on the search path
- Usually, a function will only call its arguments, which R can find in the active runtime environment
- Finally, after the function has been totally executed, the active environment is switched back to the calling environment
- Now R executes any other commands in the line of code that called the function
- If the result of the function is saved to an object with <- operator, the new object will be stored in the calling environment

# For Loop

Syntax:

```
for (value in c("I", "study", "in", "HITK")){
 print(value)
}
```

- The ‘value’ symbol in for loop acts like an argument in a function
- The for loop creates an object named ‘value’ and assigns it a new value on each run of the loop, until all of the elements have been assigned to ‘value’
- So in the above code, the print statement is executed 4 times with different values of ‘value’ and displays the following:

```
"I"
"study"
"in"
"HITK"
```

- After the execution of the for loop, the object ‘value’ will contain “HITK”, the last element
- R’s for loops execute on members of a set, not sequences of integers
- R runs the loop in whichever environment, it is called from. So the for loops overwrite the existing objects with the objects it creates

# For Loop

- Shortcoming of for loop: for loops do not return output
- The for loop must be written such that it saves its own output as it runs

```
words <- vector(length = 4) # create an empty vector
cnt <- 1 # initialize cnt
for (value in c("I", "study", "in", "HITK")){
 words[cnt] <- value
 cnt <- cnt + 1
}
```

- words will display      “I”      “study”   “in”      “HITK”
- The following code will do the same thing

```
words <- vector(length = 4)
ch <- c("I", "study", "in", "HITK")
for (cnt in 1:4){
 words[cnt] <- ch[cnt]
}
```

# While Loop

- A while loop reruns a chunk of code, while a certain condition remains TRUE
- The syntax of while loop is:

```
while (condition) {
 chunk of code
}
```

- If condition evaluates to TRUE, the code between the braces will be run. If condition evaluates to FALSE, the loop will be finished
- If the code has no relationship with the condition, the loop will run indefinitely. This is called infinite loop
- Like for loops, while loops also do not return the results
- While loop is normally used to iterate a code for a varying number of times. Example

```
while (! EOF) {
 Read next line of a file
}
```

# Repeat Loop

- Repeat loop will repeat a chunk of code until the execution is stopped (by hitting Escape) or until the command ‘break’ is encountered
- The syntax of repeat loop is:

```
repeat {
 chunk of code
 if(condition){
 break
 }
}
```

- In the repeat loop, the chunk of code will be executed at least once as the condition is normally checked at the end of the loop
- In a while loop, the chunk of code may not be executed at all, as the condition is checked at the beginning

# Debugging R Code

# Debugging R Code

- R has a simple set of debugging tools
- These tools are used to better understand the code which produces error or returns an unexpected result (bug in the code)
- Besides the user code, the functions in R or any of its packages may also be examined by the debugging tools
- R's debugging tools include traceback, browser, debug, debugonce, trace, and recover functions
- Using these tools is usually a two-step process:
  - Locate where an error has occurred
  - Try to determine why it occurred

# Debugging R Code: traceback

- The ‘traceback’ tool pinpoints the location of an error
  - Many R functions call other R functions, which call other functions, and so on
  - When an error occurs, it is not clear which of these functions went wrong
  - By typing `traceback()` at the command line, one can see the path of functions called, before an error was hit
  - ‘traceback’ will return a call stack, with the bottom function as the command entered in the command line and the top function as the one which caused error
  - ‘traceback’ will always refer to the last error encountered
  - To trace back the earlier errors, one should recreate it before running ‘traceback’
  - Example:
    - `first <- function() second()`
    - `second <- function() third()`
    - `third <- function() fourth()`
    - `fourth <- function() bug()`
    - `first()` will display Error in `bug()` : could not find function "bug"
- traceback() will display the following:
- 4: `fourth()` at #1  
3: `third()` at #1  
2: `second()` at #1  
1: `first()`

# Debugging R Code: traceback

- Advantages of traceback
  - It returns a list of suspects. Each function is more suspicious than the ones below it
  - It can show whether the paths actually followed are as per expectations
  - It can reveal whether there is any infinite recursion.
    - For example, if fourth calls second (in place of bug), then there will be infinite recursion
    - When first() is called, after a while, it will be noticed that the code is repeating itself and will return the following error

Error: C stack usage 15927440 is too close to the limit
    - ‘traceback’ will show
      - 1000: fourth() at #1
      - 999: third() at #1
      - 998: second() at #1
      - 997: fourth() at #1
      - 996: third() at #1
      - 995: second() at #1
      - 994: fourth() at #1
      - .....
- Whenever an error occurs, RStudio displays it in a gray box with an option “Show Traceback”
- If it is clicked (not necessary to type traceback() in command line), the traceback call stack is displayed

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: Computing with R  
Lecture 32: 25/01/2022

Dr. Debranjan Sarkar

# Debugging R Code: browser

- A call to `browser()` in the body of a function will pause in the middle of running the function and give control back to the user
- This allows the user to enter new commands at the command line
- The active environment for these new commands will be the runtime environment of the function which was paused, and not the global environment, as usual
- This allows the user to
  - Look at the objects, being used by the function
  - Look up their values with the same scoping rules that the function would use
  - Run code under the same conditions that the function would run it in

# Debugging R Code: browser

- Example:

```
score <- function(symbols){
 same <- symbols[1] == symbols[2] && symbols[2] == symbols[3]
 great <- symbols[1] > symbols[2] && symbols[2] > symbols[3]
 if(same) {
 prize <- 10
 }else if(great) {
 prize <- 8
 } else {
 prize <- 5
 }
 browser()
 prize <- prize * 100
}
```

# Debugging R Code: browser

```
> score(c("K", "K", "K"))
```

```
called from: score(c("K", "K", "K"))
```

```
Browse[1]> symbols
```

```
[1] "K" "K" "K"
```

```
Browse[1]> same
```

```
[1] True
```

```
Browse[1]> great
```

```
[1] False
```

```
Browse[1]> Q
```

```
>
```

# Debugging R Code: browser

- When score function is run, while executing first few statements, it come across the call to browser() and runs it
- When browser() function is run, the following things happen:
  - R stops running 'score'
  - Command prompt changes to Browse[1]> and Control is given back to the user. New commands at the new command prompt may be typed
  - Three buttons (Next, Continue and Stop) will appear above the console pane
  - RStudio will display the source code for 'score' in the scripts pane and the line containing the call to browser will be highlighted
  - The environments tab will reveal the objects which are saved in the runtime environment of 'score', instead of revealing the objects that are saved in the global environment
  - RStudio will open a new Traceback pane, which shows the call stack RStudio took to get to the browser. The most recent function ('score') will be highlighted

# Debugging R Code: browser

- The new mode, when the ‘browser’ function is running, is called browser mode
- This mode helps the user to uncover bugs
- Any command, run in the browser mode, will be evaluated in the context of the runtime environment of the function that called browser
- This is the function (here ‘score’) that is highlighted in the new traceback pane
- While we are in browser mode, the active environment will be score’s runtime environment
- This allows us to
  - Inspect the objects that ‘score’ uses
  - One can run the code and see the same results that ‘score’ would see

# Debugging R Code: browser

- Inspect the objects that ‘score’ uses
  - The updated Environments pane shows the objects saved by ‘score’ in its local environment
  - One can inspect these objects by typing their name at the browser prompt
  - Thus the values of the variables may be seen at runtime (normally not possible)
  - If a value is clearly wrong, the bug can be found out
- One can run the code and see the same results that ‘score’ would see
  - The remaining lines of the ‘score’ function can be run to see if they do anything unusual
  - These lines may be run by
    - Typing them in the command prompt, or
    - Using the three navigation buttons (Next, Continue, Stop), which now appear above the prompt
      - Next button will run the next line (or next code chunk like a ‘for’ loop or ‘if’ tree etc.) of the ‘score’
      - Continue button will run all the remaining lines of ‘score’ and then exit the browser mode
      - Stop button will exit browser mode without running any more lines of ‘score’

# Debugging R Code: browser

- Same things can be done by typing the commands n, c, Q respectively in the browser prompt
- To look up the objects with names n, c, and Q, one has to use the commands get("n"), get("c"), and get("Q") respectively
- Any object can be looked up by using the get command
- 'cont' is a synonym of c (i.e. continue) and 'where' prints the call stack
- So to look up the object with name cont or where, one has to use get command
- Browser mode can help see things from the perspective of the functions, but it cannot show where the bug lies
- Browser mode can help investigate function behaviour, which is required to spot and fix a bug
- The browser mode is the basic debugging tool of R
- Once the bug is fixed, the function without browser call has to be saved so that it does not unnecessarily pause whenever the function is called

# Debugging R Code: Break Points

- One can add a browser statement to a function by providing Break Points in a graphical way, as follows:
  - Open the script, where the function was defined
  - Click to the left of the line number of the code, where the browser statement is to be added
  - A hollow red dot will appear to show where the break point will occur
  - Run the script by clicking the Source button at the top of the Scripts pane
  - The hollow dot will turn into a solid red dot to show that the function has a breakpoint there
  - R will treat the break point like a browser statement and will enter browser mode when this is encountered
  - A break point may be removed by clicking on the red dot
  - The dot will disappear and the break point will be removed
- Break points and the browser call provide a useful way to debug functions that have been defined by the user

# Debugging R Code: debug and debugonce

- To add a browser call to the very start of a pre-existing function in R, run the function ‘debug’ with argument as the name of the pre-existing function
- One can run debug on ‘sample’ as `debug(sample)`
- Afterwards, when the function sample is called, R will act as if there is a `browser()` statement at the very beginning of the function sample
- When the function is run, it immediately enters browser mode and allows the user to step through the function one line at a time
- This will be continued until the browser statement is removed with `undebug(sample)`
- Whether a function is in ‘debugging’ mode can be checked by `isdebugged(sample)`
- If `debugonce(sample)` is used in place of `debug(sample)`, it enters the browser mode only once and automatically undebugged after the execution of the function
- In case of requirement, ‘debugonce’ may have to be called again
- Whenever an error occurs, RStudio displays it in a gray box with an option “Rerun with debug” beneath “Show Traceback”
- If “Rerun with debug” is clicked, RStudio will rerun the command, as if first ‘debugonce’ was run on this
- R will immediately go into the browser mode, allowing the user to step through the code
- The browser behaviour will only occur on this run of the code and it is not necessary to call `undebug` at the end

# Debugging R Code: trace

- ‘trace’ function is used to add the browser statement further into a function (may not be at the very beginning)
- `trace("sample", browser, at = 2)` will insert a call to browser at the second line of the function ‘sample’
- One can use the trace to insert other R functions (not just browser) into a function
- ‘trace’ can also be run on a function without inserting any new code
- If `trace(print)` is called, R will display `trace: print` at the command line, every time R runs the `print` function
- To revert back, ‘untrace’ with argument as name of the function, is to be called

# Debugging R Code: recover

- ‘recover’ function combines the call stack of traceback with the browser mode of browser
- ‘recover’ may be used, just like ‘browser’, by inserting it directly into a function’s body
- Suppose that the function ‘fourth’ calls ‘recover’ instead of ‘bug’
- fourth <- function() recover()
- When R runs ‘recover’,
  - It will pause and display the call stack upside down unlike traceback
  - It gives an option of opening a browser mode in any of the functions that appear in the call stack

```
> first()
```

Enter a frame number, or 0 to exit

```
1: first()
```

```
2: # 1: second()
```

```
3: # 1: third()
```

```
4: # 1: fourth()
```

- To enter a browser mode, type in the frame number or to exit type 0
- ‘recover’ gives a chance to inspect variables up and down the call stack and is a powerful tool for debugging
- If the code ‘options(error = recover)’ is run, R will automatically call recover() whenever there is an error
- This behaviour will last until the R session is closed, or the code ‘options(error = NULL)’ is run

# Function system.time()

- Function `system.time()` takes as argument, some R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression
- `system.time()` internally calls the function `proc.time()`, evaluates expression and then calls `proc.time()` once again
- `system.time()` finally returns the difference between the two `proc.time` calls
- `proc.time()` determines how much real and CPU time (in seconds) the currently running R process has already taken
- `system.time()` computes the time (in seconds) needed to execute an expression
  - If there is an error, gives time until the error occurred
- `system.time()` returns an object of class “`proc_time`”
  - User time: Time charged to the CPU(s) for the execution of the user instructions (expression)
  - System time: Time charged to the CPU(s) for execution by the system on behalf of the calling process
  - Elapsed time: wall clock time

# Using system.time()

Example:

```
system.time({
 for (i in 1:10000000){
 j = i * 2
 }
}
user system elapsed
0.71 0.00 0.72
```

- For straight computing tasks, the user time and the elapsed time are normally very close
- Elapsed time may be greater than user time, if the CPU spends a lot of time waiting around
- Elapsed time may be smaller than user time if the computer has multiple cores/processors (and is capable of using them)

# Beyond system.time()

- `system.time()` allows a user to check whether certain function or code blocks take excessive amount of time
- If it is already known where the problem is, one can call `system.time()` on it
- But if it is not known where the problem is, profiling of R code is necessary

Thank you

# Introduction to Data Analysis with Python and R (CSEN 3135)

Module 4: Computing with R  
Lecture 33: 31/01/2022

Dr. Debranjan Sarkar

# Profiling R Code

- After a code (in R) is developed, it might work well but it might be very slow
- Often the code runs fine for once, but if the code is to be put in a loop for 1000 iterations, it might slow down drastically
- Sometimes, it is obvious where the bottleneck lies, sometimes it is a guess, relying on intuition and on wisdom from the broader R community about speeding up R code
- However, it can lead to a focus on optimizing things that actually take a small proportion of the overall running time and might lead to wastage of time to optimize a wrong piece of code
- For example, to make a loop run 5 times faster sounds like a huge improvement. But if that loop takes 10% of the total time, it is only 8% overall speedup
- To make slow code run faster, it is important to have accurate information about which part of the code contributes to the slowness and how much time each function is taking
- This is the purpose of [Code Profiler](#), a tool for helping users to examine, in a systematic way, how much time is spent in different parts of a program
- Code profiling (better than guessing) can inform where the code spends most of its time and identify key bottlenecks in the code
- This cannot be done without performance analysis or profiling

# Profiling R code

- Profiling is a program analysis method for determining where a program spends most of its execution time
- It is very helpful in guiding programmer to improve program performance
- R includes a sampling-based profiling mechanism that records information about calls on the stack at specified time intervals
- If available, information about the specific source code lines active at the sampling point is also recorded
- Information about time spent in the garbage collector can also be collected
- Garbage collection (GC) refers to the automatic release of memory when an object is no longer used
- The collected profiling data is written to a file, by default the file Rprof.out in the current working directory
- It is good to break a code into functions so that the profiler can give useful information about where time is being spent

# Rprof() function

- Convenient functions for profiling an R code are `Rprof()` and `summaryRprof()`
- `Rprof()` function is a built-in tool which enables or disables profiling of the execution of the R expressions
- If profiling is enabled, the profiler stops the R interpreter at regular time intervals, records the current function call stack, and saves the information in a file
- The results from `Rprof()` are stochastic. Each time we run a function R, the conditions might be changed
- Hence, each time a code is profiled, the result will be slightly different
- `Rprof()` keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function
- By default, the profiler samples the function call stack every 0.02 seconds
- If a code runs very fast (say, less than 0.02 seconds), the profiler is not useful

# Rprof() function

- Syntax: Rprof(filename, append = FALSE, interval = 0.02)
- Many other arguments like memory.profiling, gc.profiling, line.profiling, etc.
- The argument ‘filename’ indicates the name of the file to be used for recording the profiling results. To disable profiling, the filename is put to NULL or “”
- The logical argument ‘append’ indicates whether the file be overwritten or appended to (defauLt is FALSE => overwritten)
- The Real argument ‘interval’ indicates the time interval between samples (by default 0.02 second or 20 msec)
- memory.profiling is a logical argument indicating whether to write memory use information in the file
- gc.profiling is a logical argument indicating whether to record whether GC is running
- line.profiling is a logical argument showing whether to write line locations in the file
- Enabling profiling automatically disables any existing profiling to another or the same

# Summarizing profiling data

- `Rprof()` creates a file to record the profiling results
- Profiling writes the call stack every ‘interval’ seconds, to the file specified
- However, this file is not directly readable
- We have to use some function or package or tools to process the output file created by the `Rprof()` function, to produce a summary of the usage
- The `summaryRprof()` function summarizes the output from `Rprof()`
- This function is available in the base package in R and provides a simple interface for examining the profiling data
- The `summaryRprof()` function tabulates the R profiler output and calculates how much time is spent in which function

# Summarizing profiling data

- The ‘profvis’ package provides an interactive graphical interface for visualizing code-profiling data from Rprof()
- It is required to install ‘profvis’ from CRAN (Comprehensive R Archive Network)
- The ‘proftools’ package provides a much more extensive set of tools for summarizing, visualizing, and filtering this data
- We shall discuss on summaryRprof() function

# Using summaryRprof()

- Syntax: `summaryRprof(filename)`
- filename is the name of the file produced by Rprof
- There are many other arguments of the function
- When default values for these arguments are used, we get the following
  - A data frame of timings sorted by ‘self’ time (`by.self`)
  - A data frame of timings sorted by ‘total’ time (`by.total`)
  - `sampling.interval` showing the interval of sampling (by default 0.02 second)
  - `sampling.time` showing total time of profiling run

# Using summaryRprof()

- The first two data frames have columns self.time, self.pct, total.time and total.pct
- self.time: how many seconds were spent in that function only
- total.time: how many seconds were spent in that function including the functions that it called
- self.pct: what percent is the self time of a function w.r.t. the overall time. (These numbers should add to 100%)
- total.pct: what percentage is the total time of a function w.r.t. the overall time (These numbers will typically add to greater than 100%, because functions will be counted for themselves, and for all the functions that call them)

# Using Rprof() and summaryRprof(): Example

```
t <- function(i){
 i +10
}
x <- function() {
 for (i in 1:1000000){
 a <- t(i)
 }
 return(a)
}
> Rprof (tmp <- "rprof.out")
> x()
> Rprof(NULL)
> summaryRprof(tmp)
```

# Result of summaryRprof()

\$by.self

|               | self.time | self.pct | total.time | total.pct |
|---------------|-----------|----------|------------|-----------|
| "x"           | 1.44      | 91.14    | 1.58       | 100.00    |
| "t"           | 0.12      | 7.59     | 0.12       | 7.59      |
| "<Anonymous>" | 0.02      | 1.27     | 0.02       | 1.27      |

\$by.total

|               | total.time | total.pct | self.time | self.pct |
|---------------|------------|-----------|-----------|----------|
| "x"           | 1.58       | 100.00    | 1.44      | 91.14    |
| "t"           | 0.12       | 7.59      | 0.12      | 7.59     |
| "<Anonymous>" | 0.02       | 1.27      | 0.02      | 1.27     |

\$sample.interval

[1] 0.02

\$sampling.time

[1] 1.58

# Simulation with R Code

- Monte Carlo simulation consists of three steps
  - Simulate one trial of experiment
  - Determine whether success occurred
  - Replicate steps 1 and 2
- Problem: Use R to simulate the probability of getting (at least one) 8 in the sum of two dice rolled
- trials <- 10000
- die1 <- sample(1:6, size = trials, replace = TRUE)
- die2 <- sample(1:6, size = trials, replace = TRUE)
- result <- die1 + die2 # get a vector of 10000 results
- prob <- mean(result == 8) # give the number of 8 out of the total number (i.e. 10000)
- When trials is assigned a value of 10000, prob will display 0.1291
- When trials is assigned a value of 100000, prob will display 0.13891
- Theoretical result is  $(5 / 36) = 0.138888888$

# Some useful functions in R

- Built-in functions in R may be categorized as follows:
  - General
  - Mathematical
  - Graphical
  - Statistical ....

# List of some useful functions (General)

- `abs(-2)` will display 2
- `append(c(6, 7, 8), c(3,4,5), 2)` will display 6 7 3 4 5 8
- `cat("How are you?")` will display How are you?
- `date_origin = as.POSIXlt("2020-01-01")`
- `date = as.POSIXlt("2022-01-31")`
- `julian(date, date_origin)` will display Time difference of 761 days
- `ls()` shows the objects in the current environment
- `range(5, 8, 3, -1, 4, 5, 10, 7)` will display -1 10
- `rep(5, 3)` will display 5 5 5
- `seq(1, 11, 3)` will display 1 4 7 10
- `sequence(c(3, 5, 6))` will display 1 2 3 1 2 3 4 5 1 2 3 4 5 6
- `sign(-10)` will display -1

# List of some useful functions (General)

- `sort(c(5, 3, -8, 9, 12, 5, 3))` will display -8 3 3 5 5 9 12
- `sort(c(5, 3, -8, 9, 12, 5, 3), decreasing = TRUE)` 12 9 5 5 3 3 -8
- `tolower("ABCDEF")` will display "abcdef"
- `toupper("bjiijhnjf67346i")` will display "BJIIJHNJF67346I"
- `unique(c(2, 45, 2, 7, 4, 9, 4, 2, 9))` will display 2 45 7 4 9
- `floor(3.7)` will display 3
- `ceiling(3.7)` will display 4
- `round(3.7)` will display 4
- `signif(3.141519, digits = 4)` will display 3.142
- `trunc(3.141519)` will display 3
- `Sys.time()` will display "2022-01-30 20:10:12 IST"

# List of some useful functions (General)

- `file.info("C:/Users/Admin/Desktop/x.docx")` will display

|                               | size  | isdir | mode | mtime               | ctime               | atime             | exe |
|-------------------------------|-------|-------|------|---------------------|---------------------|-------------------|-----|
| C:/Users/Admin/Desktop/x.docx | 14980 | FALSE | 666  | 2019-01-28 01:05:14 | 2019-01-27 23:51:24 | 19-01-28 01:05:14 | no  |

- The following information about the file "C:/Users/Admin/Desktop/x.docx" will be displayed

- Name of the file, Size in bytes
- isdir to indicate whether this is a directory or file
- mode indicates the permission, 666 (in octal) means all users can read and write but cannot execute
- mtime: time last modified
- ctime: time created
- atime: time last accessed
- exe indicates whether this is an executable file
- Built-in constants: pi, letters, LETTERS

pi => 3.141593

LETTERS[2] => "B"

letters[7] => "g"

- month.abb, month.name

month.abb[3] => "Mar"

month.name[7] => "July"

# List of some useful functions (Mathematical)

- sqrt(), sum(), log, log2, exp, cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, union, intersect, setdiff, setequal
- Integration (Function ‘integrate’)

```
integrand <- function(x) {1/((x+1)*sqrt(x))}
```

integrate(integrand, lower = 0, upper = Inf) will display

3.141593 with absolute error < 2.7e-05

```
integrand = function(x) cos(x)
```

integrate(integrand, lower = 0, upper = pi/2) will display

1 with absolute error < 1.1e-14

# List of some useful functions (Mathematical)

- Differentiation (Function 'deriv')

```
ddx <- deriv(~x^3, "x")
```

mode(ddx) will display "expression"

```
x = 4
```

eval(ddx) will display the following:

```
[1] 64
```

```
attr(,"gradient")
```

```
x
```

```
[1,] 48
```

# List of some useful functions (Graphical)

- `plot(sin, 0, 2*pi)`
- `plot(1:100, (1:100) ^ 2, main = "plot(1:100, (1:100) ^ 2)")`
- `curve(sin, -2*pi, 2*pi, xname = "t")`
- `curve(x^3 - 3*x, -2, 2)`
- `curve(x^2 - 2, add = TRUE, col = "violet")`
- `points(c(-2, -1, 0, 1, 2), c(-2, -1, 0, 1, 2), col = "red")`
- `plot(0:100, 0:100, main = "arrows")`
- `arrows(40, 20, 60, 20)`
- `lines(c(20, 30, 50), c(40, 20, 90))`
- `segments(50, 90, 60, 100, col= 'red')`
- `matplot((-4:5)^2, main = "Quadratic")`

Thank you