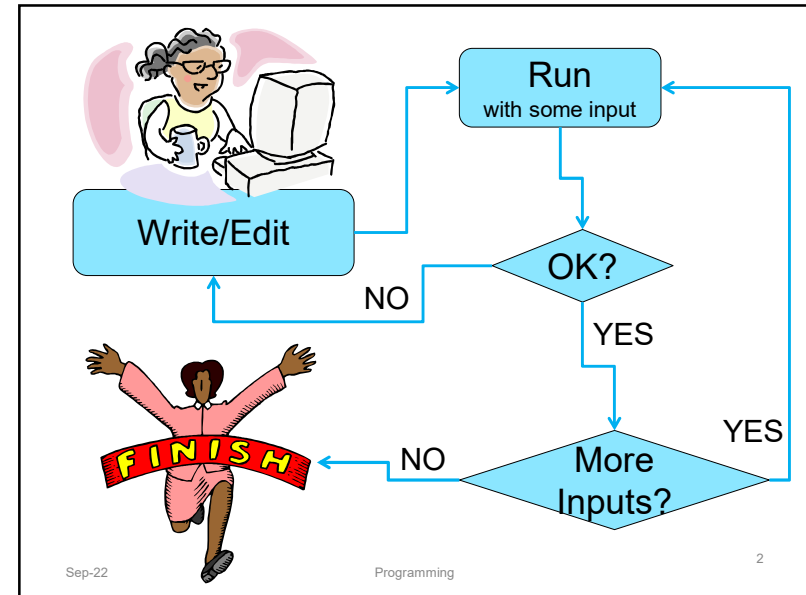


# Python Programming

Welcome

Python Programming

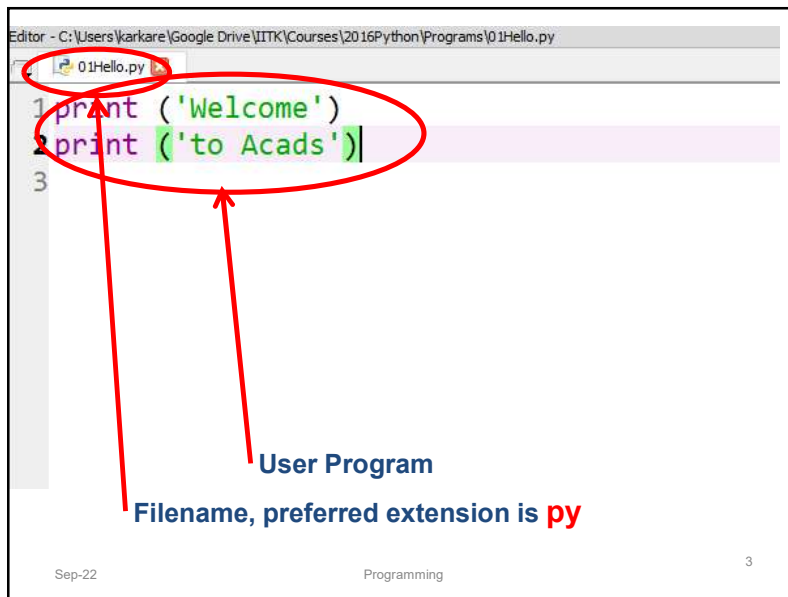
1



Sep-22

Programming

2



Sep-22

Programming

3

**IN[1] :** ← **Python Shell Prompt**  
 Welcome  
 to Acads

**IN[2] :**  
 8

**IN[3] :** 3 > 5  
 False

**IN[4] :** print ('3 + 5 is', 3 + 5)  
 3 + 5 is 8

← **User Commands (Statements)**

← **Outputs**

**Python Shell is Interactive**

Sep-22

Programming

4

## Interacting with Python Programs

- Python program communicates its results to user using **print**
- Most useful programs require information from users
  - Name and age for a travel reservation system
- Python 3 uses **input** to read user input as a string (**str**)

Sep-22

Programming

5

## input

- Take as argument a **string** to print as a prompt
- Returns the user typed value as a **string**
  - details of how to process user string later

```
IN[1]: age = input('How old are you?')
How old are you?
IN[2]: print('You are', age, 'years old')
You are 35 years old
IN[3]:
<type 'str'>
```

Sep-22

Programming

6

## Elements of Python

- A Python program is a sequence of **definitions** and **commands (statements)**
- Commands manipulate **objects**
- Each object is associated with a **Type**
- **Type:**
  - A set of values
  - A set of operations on these values
- **Expressions:** An operation (combination of objects and **operators**)

Sep-22

Programming

7

## Types in Python

- **int**
  - Bounded integers, e.g. 732 or -5
- **float**
  - Real numbers, e.g. 3.14 or 2.0
- **long**
  - Long integers with unlimited precision
- **str**
  - Strings, e.g. 'hello' or 'C'

Sep-22

Programming

8

## Types in Python

### • Scalar

- Indivisible objects that do not have internal structure
- **int** (signed integers), **float** (floating point), **bool** (Boolean), **NoneType**
  - NoneType is a special type with a single value
  - The value is called **None**

### • Non-Scalar

- Objects having internal structure
- **str** (strings)

Sep-22

Programming

9

## Example of Types

```
In [14]: type(500)
Out[14]: int

In [15]: type(-200)
Out[15]: int

In [16]: type(3.1413)
Out[16]: float

In [17]: type(True)
Out[17]: bool

In [18]: type('Hello Class')
Out[18]: str

In [19]: type(3!=2)
Out[19]: bool
```

Sep-22

Programming

10

## Type Conversion (Type Cast)

- Conversion of value of one type to other
- We are used to **int ↔ float** conversion in Math
  - Integer 3 is treated as float 3.0 when a real number is expected
  - Float 3.6 is truncated as 3, or rounded off as 4 for integer contexts
- Type names are used as type converter functions

Sep-22

Programming

11

## Type Conversion Examples

```
In [20]: int(2.5)
Out[20]: 2

In [21]: int(2.3)
Out[21]: 2

In [22]: int(3.9)
Out[22]: 3

In [23]: float(3)
Out[23]: 3.0

In [24]: int('73')
Out[24]: 73

In [25]: int('Acads')
Traceback (most recent call last):

  File "<ipython-input-25-90ec37205222>", line 1, in <module>
    int('Acads')

ValueError: invalid literal for int() with base 10: 'Acads'
```

Note that float to int conversion is truncation, not rounding off

```
In [26]: str(3.14)
Out[26]: '3.14'

In [27]: str(26000)
Out[27]: '26000'
```

Sep-22

Programming

12

## Type Conversion and Input

```
In [11]: age = input('How old are you? ')
```

How old are you? 35

```
In [12]: print('In 5 years, your age will be', age + 5)
Traceback (most recent call last):
```

```
File "<ipython-input-12-7fb7a9e926c2>", line 1, in <module>
    print('In 5 years, your age will be', age + 5)
```

**TypeError: Can't convert 'int' object to str implicitly**

```
In [13]: print('In 5 years, your age will be', int(age) + 5)
In 5 years, your age will be 40
```

Sep-22

Programming

13

## Operators

- Arithmetic

+	-	*	//	/	%	**
---	---	---	----	---	---	----

- Comparison

==	!=	>	<	>=	<=
----	----	---	---	----	----

- Assignment

=	+=	-=	*=	//=	/=	%=	**=
---	----	----	----	-----	----	----	-----

- Logical

and	or	not
-----	----	-----

- Bitwise

&		^	~	>>	<<
---	--	---	---	----	----

- Membership

in	not in
----	--------

- Identity

is	is not
----	--------

Sep-22

Programming

14

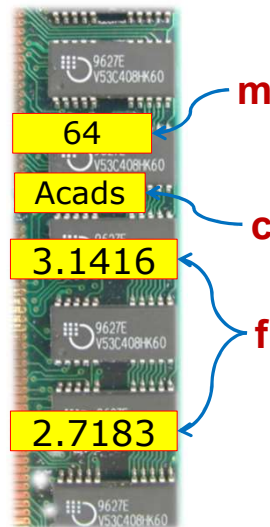
## Variables

- A name associated with an object
- Assignment used for binding

```
m = 64;
c = 'Acads';
f = 3.1416;
```

- Variables can change their bindings

```
f = 2.7183;
```



Sep-22

Programming

15

## Assignment Statement

- A simple assignment statement  
*Variable = Expression;*
- Computes the value (object) of the expression on the right hand side expression (**RHS**)
- Associates the name (variable) on the left hand side (**LHS**) with the RHS value
- =** is known as the assignment operator.

Sep-22

Programming

16

## Multiple Assignments

- Python allows multiple assignments

`x, y = 10, 20` Binds x to 10 and y to 20

- Evaluation of multiple assignment statement:

- All the expressions on the RHS of the `=` are first evaluated **before any binding happens**.
- Values of the expressions are bound to the corresponding variable on the LHS.

`x, y = 10, 20`  
`x, y = y+1, x+1`  
 x is bound to 21 and y to 11 at the end of the program

Sep-22

Programming

## Programming using Python

### Operators and Expressions

9/27/2022

Programming

18

## Binary Operations

Op	Meaning	Example	Remarks
+	Addition	9+2 is 11 9.1+2.0 is 11.1	
-	Subtraction	9-2 is 7 9.1-2.0 is 7.1	
*	Multiplication	9*2 is 18 9.1*2.0 is 18.2	
/	Division	9/2 is 4.5 9.1/2.0 is 4.55	In Python3 Real div.
//	Integer Division	9//2 is 4	
%	Remainder	9%2 is 1	

9/27/2022

Programming

19

## The // operator

- Also referred to as “integer division”
- Result is a whole integer (floor of real division)
  - But the type need not be `int`
  - the integral part of the real division
  - rounded towards minus infinity ( $-\infty$ )

- Examples

<code>9//4 is 2</code>	<code>(-1)//2 is -1</code>	<code>(-1)//(-2) is 0</code>
<code>1//2 is 0</code>	<code>1//(-2) is -1</code>	<code>9//4.5 is 2.0</code>

9/27/2022

Programming

20

## The % operator

- The remainder operator **%** returns the remainder of the result of dividing its first operand by its second.

<b>9%4 is 1</b>	<b>(-1)%2 is 1</b>	<b>(-1)//(-2) is 0</b>
<b>9%4.5 is 0.0</b>	<b>1%(-2) is 1</b>	<b>1%0.6 is 0.4</b>

**Ideally:  $x == (x//y)*y + x \%y$**

9/27/2022

Programming

21

## Conditional Statements

- In daily routine
  - If it is very hot, I will skip exercise.
  - If there is a quiz tomorrow, I will first study and then sleep. Otherwise I will sleep now.
  - If I have to buy coffee, I will go left. Else I will go straight.



Sep-22

Programming

22

## if-else statement

- Compare two integers and print the min.

```
if x < y:
    print (x)
else:
    print (y)
    print ('is the minimum')
```

1. Check if x is less than y.
2. If so, print x
3. Otherwise, print y.

Sep-22

Programming

23

## Indentation

- Indentation is **important** in Python
  - grouping of statement (block of statements)
  - no explicit brackets, e.g. { }, to group statements

```
→ x,y = 6,10
→ if x < y:
→   print (x)
→ else:
→   print (y)
→   print ('is the min')
```

Run the program

6

10

Output

6

Sep-22

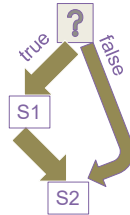
Programming

24

## if statement (no else!)

- General form of the if statement

```
if boolean-expr :
    S1
S2
```



- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to the S2.
  - If expression evaluates to **false**, then control moves to the S2 directly.

Sep-22

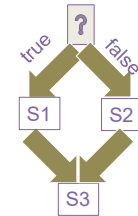
Programming

25

## if-else statement

- General form of the if-else statement

```
if boolean-expr :
    S1
else:
    S2
S3
```



- Execution of if-else statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to S3.
  - If expression evaluates to **false**, then S2 is executed and then control moves to S3.
  - S1/S2 can be **blocks of statements**!

Sep-22

Programming

26

## Nested if, if-else

```
if a <= b:
    if a <= c:
        ...
    else:
        ...
else:
    if b <= c):
        ...
    else:
        ...
```

Sep-22

Programming

27

## Elif

- A special kind of nesting is the chain of if-else-if-else-... statements
- Can be written elegantly using if-elif-...-else

```
if cond1:
    s1
else:
    if cond2:
        s2
    else:
        if cond3:
            s3
        else:
            ...
```

Sep-22

Programming

```
if cond1:
    s1
elif cond2:
    s2
elif cond3:
    s3
elif ...
else
    last-block-of-stmt
```

28

## Summary of if, if-else

- if-else, nested if's, elif.
- Multiple ways to solve a problem
  - issues of readability, maintainability
  - and efficiency

Sep-22

Programming

29

## Class Quiz

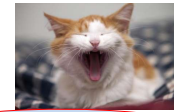
- What is the value of expression:

$(5 < 2) \text{ and } (3/0 > 1)$

- a) Run time crash/error



- b) I don't know / I don't care



- c) False

- d) True



The correct answer is **False**

Sep-22

Programming

30

## Short-circuit Evaluation

- Do not evaluate the second operand of binary short-circuit logical operator if the result can be deduced from the first operand
  - Also applies to nested logical operators

$\text{not}((2 > 5) \text{ and } (3/0 > 1)) \text{ or } (4/0 < 2)$

Evaluates to true

Sep-22

Programming

31

## 3 Factors for Expr Evaluation

### • Precedence

- Applied to two different class of operators
- $+$  and  $*$ ,  $-$  and  $*$ , **and** and **or**, ...

### • Associativity

- Applied to operators of same class
- $*$  and  $*$ ,  $+$  and  $-$ ,  $*$  and  $/$ , ...

### • Order

- Precedence and associativity **identify the operands** for each operator
- **Not which operand is evaluated first**
- Python evaluates expressions from left to right
- While evaluating an assignment, the right-hand side is evaluated before the left-hand side.

Sep-22

Programming

32



## Class Quiz

- What is the output of the following program:

```
y = 0.1*3
if y != 0.3:
    print ('Launch a Missile')
else:
    print ("Let's have peace")
```

Launch a Missile

## Caution about Using Floats

- Representation of *real numbers* in a computer can not be exact
  - Computers have limited memory to store data
  - Between any two distinct real numbers, there are infinitely many real numbers.
- On a typical machine running Python, there are 53 bits of precision available for a Python float

## Caution about Using Floats

- The value stored internally for the decimal number 0.1 is the binary fraction  
 $0.00011001100110011001100110011001100110011001100110011010$
- Equivalent to decimal value  
 $0.1000000000000000055511151231257827021181583404541015625$
- Approximation is similar to decimal approximation  $1/3 = 0.33333333...$
- No matter how many digits you use, you have an approximation

## Comparing Floats

- Because of the approximations, comparison of floats is not exact.
- Solution?**
- Instead of

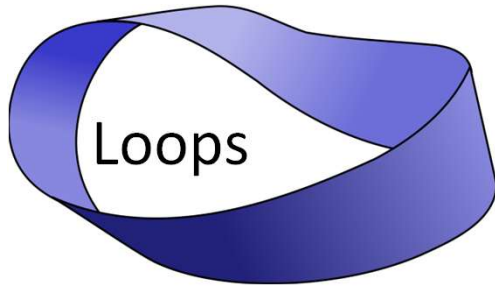
$x == y$

use

$\text{abs}(x-y) \leq \text{epsilon}$

where **epsilon** is a suitably chosen small value

## Programming using Python



Sep-22

Python Programming

37

## Printing Multiplication Table

5	X	1	=	5
5	X	2	=	10
5	X	3	=	15
5	X	4	=	20
5	X	5	=	25
5	X	6	=	30
5	X	7	=	35
5	X	8	=	40
5	X	9	=	45
5	X	10	=	50

Sep-22

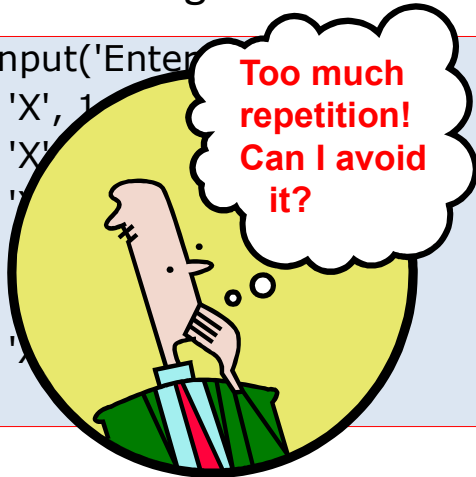
Python Programming

38

## Program...

```
n = int(input('Enter n: '))
print (n, 'X', 1)
print (n, 'X', 2)
print (n, 'X', 3)
print (n, 'X', 4)
print (n, 'X', 5)
print (n, 'X', 6)
print (n, 'X', 7)
print (n, 'X', 8)
print (n, 'X', 9)
print (n, 'X', 10)
....
```

**Too much  
repetition!  
Can I avoid  
it?**

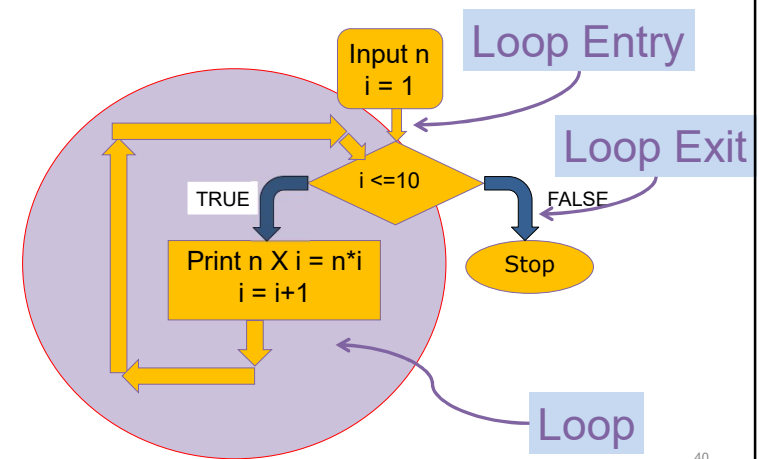


Sep-22

Python Programming

39

## Printing Multiplication Table

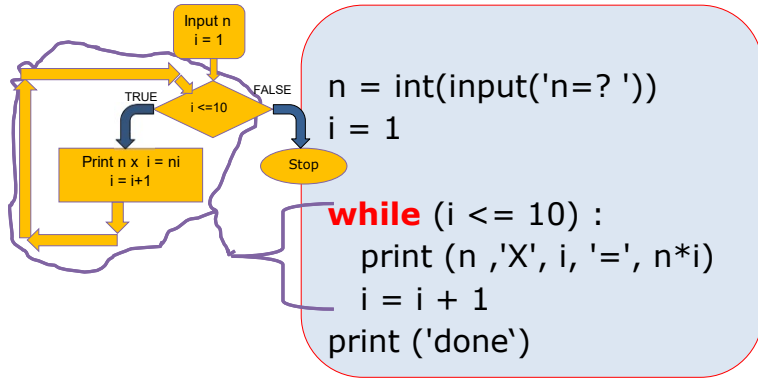


Sep-22

Python Programming

40

## Printing Multiplication Table



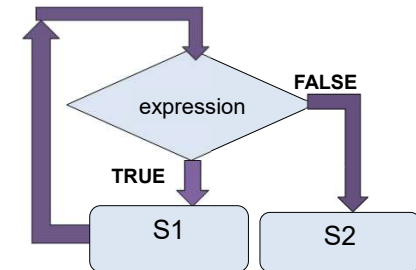
Sep-22

Python Programming

41

## While Statement

```
while (expression):
    S1
    S2
```



1. Evaluate expression
2. If TRUE then
  - a) execute statement1
  - b) goto step 1.
3. If FALSE then execute statement2.

Sep-22

Python Programming

42

## For Loop

- Print the sum of the reciprocals of the first 100 natural numbers.

```
rsum=0.0# the reciprocal sum

# the for loop
for i in range(1,101):
    rsum = rsum + 1.0/i
print ('sum is', rsum)
```

Sep-22

Python Programming

43

## For loop in Python

- General form

```
for variable in sequence:
    stmt
```

Sep-22

Python Programming

44

## range

- `range(s, e, d)`  
– generates the list:  
 $[s, s+d, s+2*d, \dots, s+k*d]$   
where  $s+k*d < e \leq s+(k+1)*d$
- `range(s, e)` is equivalent to `range(s, e, 1)`
- `range(e)` is equivalent to `range(0, e)`

**Exercise:** What if `d` is negative? Use python interpreter to find out.

Sep-22

Python Programming

45

## Quiz

- What will be the output of the following program

```
# print all odd numbers < 10
i = 1
while i <= 10:
    if i%2==0: # even
        continue
    print (i, end=' ')
    i = i+1
```

Sep-22

Python Programming

46

## Continue and Update Expr

- Make sure `continue` does not bypass update-expression for while loops



```
# print all odd numbers < 10
i = 1
while i <= 10:
    if i%2==0: # even
        continue
    print (i, end=' ')
    i = i+1
```

*i is not incremented  
when even number  
encountered.  
Infinite loop!!*

Sep-22

Python Programming

47

## Programming using Python

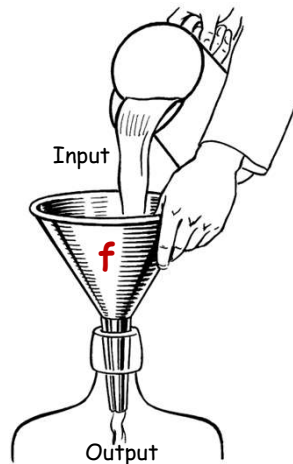
# f(unctions)

Sep-22

Programming, Functions

48

## Parts of a function



Sep-22

49

```
def max(a, b):
    "return maximum among a and b"
    if (a > b):
        return a
    else:
        return b
```

**keyword**

**Function Name**

**2 arguments a and b (formal args)**

**Body of the function, indented w.r.t the def keyword**

**Documentation comment (docstring), type help <function-name> on prompt to get help for the function**

**Call to the function. Actual args are 6 and 4.**

`x = max(6, 4)`

Sep-22

```
def max(a, b):
    "return maximum among a and b"
    if (a > b):
        return a
    else:
        return b
```

```
In[3]: help(max)
Help on function max in module __main__:

max(a, b)
    return maximum among a and b
```

Sep-22

Programming, Functions

51

## Keyword Arguments

```
def printName(first, last, initials):
    if initials:
        print (first[0] + '.' + last[0] + '.')
    else:
        print (first, last)
```

Note use of [0] to get the first character of a string. More on this later.

Call	Output
<code>printName('Acads', 'Institute', False)</code>	Acads Institute
<code>printName('Acads', 'Institute', True)</code>	A. I.
<code>printName(last='Institute', initials=False, first='Acads')</code>	Acads Institute
<code>printName('Acads', initials=True, last='Institute')</code>	A. I.

Sep-22

Programming, Functions

52

## Keyword Arguments

- Parameter passing where formal is bound to actual using formal's name
- Can mix keyword and non-keyword arguments
  - All non-keyword arguments precede keyword arguments in the call
  - Non-keyword arguments are matched by position (order is important)
  - Order of keyword arguments is not important

Sep-22

Programming, Functions

53

## Default Values

```
def printName(first, last, initials=False) :
    if initials:
        print (first[0] + ' ' + last[0] + '!')
    else:
        print (first, last)
```

Note the use  
of "default"  
value

Call	Output
<code>printName('Acads', 'Institute')</code>	Acads Institute
<code>printName(first='Acads', last='Institute', initials=True)</code>	A. I.
<code>printName(last='Institute', first='Acads')</code>	Acads Institute
<code>printName('Acads', last='Institute')</code>	Acads Institute

Sep-22

Programming, Functions

54

## Default Values

- Allows user to call a function with fewer arguments
- Useful when some argument has a fixed value for most of the calls
- All arguments with default values must be at the end of argument list
  - non-default argument can not follow default argument

Sep-22

Programming, Functions

55

## Globals

- Globals allow functions to communicate with each other indirectly
  - Without parameter passing/return value
- Convenient when two seemingly "far-apart" functions want to share data
  - No *direct* caller/callee relation
- If a function has to update a global, it must re-declare the global variable with **global** keyword.

Sep-22

Programming, Functions

56

## Globals

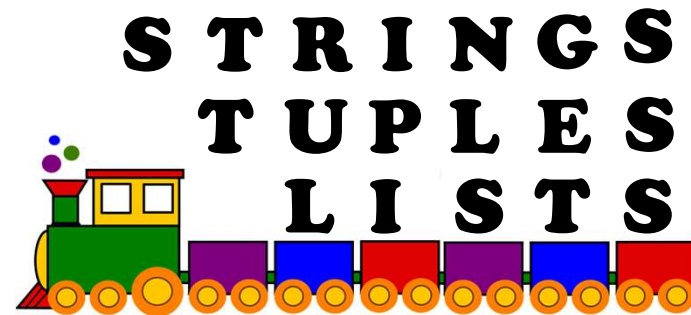
```
PI = 3.14
def perimeter(r):
    return 2 * PI * r
def area(r):
    return PI * r * r
def update_pi():
    global PI
    PI = 3.14159
```

```
>>> print(area(100))
31400.0
>>> print(perimeter(10))
62.800000000000004
>>> update_pi()
>>> print(area(100))
31415.999999999996
>>> print(perimeter(10))
62.832
```

defines **PI** to be of float type with value 3.14. **PI** can be used across functions. Any change to **PI** in **update\_pi** will be visible to all due to the use of **global**.

57

## Programming with Python



Sep-22

Programming

58

## Strings

- Strings in Python have type **str**
- They represent sequence of characters
  - Python does not have a type corresponding to character.
- Strings are enclosed in single quotes(') or double quotes("")
  - Both are equivalent
- Backslash (\) is used to escape quotes and special characters

Sep-22

Programming

59

## Strings

```
>>> name='intro to python'
>>> descr='acad\'s first course'
>>> name
'intro to python'
>>> descr
"acad's first course"
• More readable when print is used

>>> print descr
acad's first course
```

Sep-22

Programming

60

## Length of a String

- **len** function gives the length of a string

```
>>> name='intro to python'
>>> empty=''
>>> single='a'
>>> len(name)
15
>>> len(single)
1
>>> len(empty)
0
>>> special='1\n2'
>>> len(special)
```

**\n** is a **single** character:  
the special character  
representing newline

Sep-22

Programming

61

## Concatenate and Repeat

- In Python, **+** and **\*** operations have special meaning when operating on strings
  - **+** is used for concatenation of (two) strings
  - **\*** is used to repeat a string, an **int** number of time
- Function/Operator Overloading

Sep-22

Programming

62

## Concatenate and Repeat

```
>>> details = name + ', ' + descr
>>> details
"intro to python, acad's first course"
>>> print punishment
I won't fly paper airplanes in class

>>> print punishment*5
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
I won't fly paper airplanes in class
```

Sep-22

Programming

63

## Indexing

- Strings can be indexed
- First character has index 0
 

```
>>> name='Acads'
>>> name[0]
'A'
>>> name[3]
'd'
>>> 'Hello'[1]
'e'
```

Sep-22

Programming

64



## Indexing

- Negative indices start counting from the right
- Negatives indices start from -1
- -1 means last, -2 second last, ...

```
>>> name='Acads'
>>> name[-1]
's'
>>> name[-5]
'A'
>>> name[-2]
'd'
```

Sep-22

Programming

65

## Indexing

- Using an index that is too large or too small results in “index out of range” error

```
>>> name='Acads'
>>> name[50]

Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    name[50]
IndexError: string index out of range
>>> name[-50]
```

```
Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    name[-50]
IndexError: string index out of range
```

Sep-22

Programming

66

## Slicing

- To obtain a substring
- `s[start:end]` means substring of `s` starting at index `start` and ending at index `end-1`
- `s[0:len(s)]` is same as `s`
- Both `start` and `end` are optional
  - If `start` is omitted, it defaults to 0
  - If `end` is omitted, it defaults to the length of string
- `s[:]` is same as `s[0:len(s)]`, that is same as `s`

Sep-22

Programming

67

## Slicing

```
>>> name='Acads'
>>> name[0:3]
'Aca'
>>> name[:3]
'Aca'
>>> name[3:]
'ds'
>>> name[:3] + name[3:]
'Acads'
>>> name[0:len(name)]
'Acads'
>>> name[:]
'Acads'
```

Sep-22

Programming

68

## More Slicing

```
>>> name='Acads'
>>> name[-4:-1]
'cad'
>>> name[-4:]
'cads'
>>> name[-4:4]
'cad'
```

### Understanding Indices for slicing

A	c	a	d	s	
0	1	2	3	4	5
-5	-4	-3	-2	-1	

Sep-22

Programming

69

## Out of Range Slicing

A	c	a	d	s
0	1	2	3	4
-5	-4	-3	-2	-1

- Out of range indices are ignored for slicing
- when start and end have the same sign, if start >= end, empty slice is returned

Why?

```
>>> name='Acads'
>>> name[4:50]      >>> name[-50:-20]
's'                ''
>>> name[40:50]     >>> name[50:20]
''                 ''
>>> name[-50:20]    >>> name [1:-1]
'Acads'            'cad'
```

Sep-22

Programming

70

## Tuples

- A tuple consists of a number of values separated by commas

```
>>> t = 'intro to python', 'amey karkare', 101
>>> t[0]
'intro to python'
>>> t[2]
101
>>> t
('intro to python', 'amey karkare', 101)
>>> type(t)
<type 'tuple'>
```

- Empty and Singleton Tuples

```
>>> empty = ()
>>> singleton = 1, # Note the comma at the end
```

Sep-22

Programming

71

## Nested Tuples

- Tuples can be nested

```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
```

- Note that **course** tuple is copied into **student**.

– Changing **course** does not affect **student**

```
>>> course = 'Stats', 'Adam', 102
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
```

Sep-22

Programming

72

## Length of a Tuple

- len function gives the length of a tuple

```
>>> course = 'Python', 'Amey', 101
>>> student = 'Prasanna', 34, course
>>> empty = ()
>>> singleton = 1,
>>> len(empty)
0
>>> len(singleton)
1
>>> len(course)
3
>>> len(student)
3
```

73

## More Operations on Tuples

- Tuples can be concatenated, repeated, indexed and sliced

```
>>> course1
('Python', 'Amey', 101)
>>> course2
('Stats', 'Adams', 102)
>>> course1 + course2
('Python', 'Amey', 101, 'Stats', 'Adams', 102)
>>> (course1 + course2)[3]
'Stats'
>>> (course1 + course2)[2:7]
(101, 'Stats', 'Adams', 102)
>>> 2*course1
('Python', 'Amey', 101, 'Python', 'Amey', 101)
```

Sep-22

Programming

## Unpacking Sequences

- Strings and Tuples are examples of sequences
  - Indexing, slicing, concatenation, repetition operations applicable on sequences
- Sequence Unpacking operation can be applied to sequences to get the components
  - Multiple assignment statement
  - LHS and RHS must have equal length

Sep-22

Programming

75

## Unpacking Sequences

```
>>> student
('Prasanna', 34, ('Python', 'Amey', 101))
>>> name, roll, regdcourse=student
>>> name
'Prasanna'
>>> roll
34
>>> regdcourse
('Python', 'Amey', 101)
>>> x1,x2,x3,x4 = 'amey'
>>> print(x1,x2,x3,x4)
a m e y
```

Sep-22

Programming

76

## Lists

- Ordered sequence of values
- Written as a sequence of comma-separated values between square brackets
- Values can be of different types
  - usually the items all have the same type

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst
[1, 2, 3, 4, 5]
>>> type(lst)
<type 'list'>
```

Sep-22

Programming

77

## Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> fib = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> len(fib)
10
>>> fib[3] # Indexing
3
>>> fib[3:] # Slicing
[3, 5, 8, 13, 21, 34, 55]
```

Sep-22

Programming

78

## Lists

- List is also a sequence type
  - Sequence operations are applicable

```
>>> [0] + fib # Concatenation
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> 3 * [1, 1, 2] # Repetition
[1, 1, 2, 1, 1, 2, 1, 1, 2]
>>> x, y, z = [1, 1, 2] # Unpacking
>>> print(x, y, z)
1 1 2
```

Sep-22

Programming

79

## More Operations on Lists

- L.append(x)
- L.pop()
- L.extend(seq)
- L.index(x)
- L.insert(i, x)
- L.count(x)
- L.remove(x)
- L.sort()
- L.pop(i)
- L.reverse()

**x** is any value, **seq** is a sequence value (list, string, tuple, ...),  
**i** is an integer value

Sep-22

Programming

80

## Mutable and Immutable Types

- Tuples and List types look very similar
- However, there is one major difference: Lists are **mutable**
  - Contents of a list can be modified
- Tuples and Strings are **immutable**
  - Contents can not be modified

Sep-22

Programming

81

## Summary of Sequences

Operation	Meaning
<code>seq[i]</code>	i-th element of the sequence
<code>len(seq)</code>	Length of the sequence
<code>seq1 + seq2</code>	Concatenate the two sequences
<code>num*seq</code> <code>seq*num</code>	Repeat seq num times
<code>seq[start:end]</code>	slice starting from <b>start</b> , and ending at <b>end-1</b>
<code>e in seq</code>	True if e is present in seq, False otherwise
<code>e not in seq</code>	True if e is not present in seq, False otherwise
<code>for e in seq</code>	Iterate over all elements in seq (e is bound to one element per iteration)

Sequence types include String, Tuple and List.  
Lists are mutable, Tuple and Strings immutable.

Sep-22

Programming

82

## Summary of Sequences

- For details and many useful functions, refer to:  
<https://docs.python.org/3.2/tutorial/datastructures.html>

Sep-22

Programming

83

## Programming with Python

### Sets and Dictionaries

Sep-22

Esc101, Strings

84

## Sets

- An unordered collection with no duplicate elements
- Supports
  - membership testing
  - eliminating duplicate entries
  - Set operations: union, intersection, difference, and symmetric difference.

Sep-22

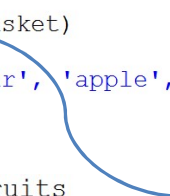
Programming

85

## Sets

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruits = set(basket)
>>> fruits
{'orange', 'pear', 'apple', 'banana'}
>>> type(fruits)
set

>>> 'apple' in fruits
True
>>> 'mango' in fruits
False
```



Create a set from a sequence

Sep-22

Programming

86

## Set Operations

```
>>> A=set('acads')
>>> B=set('institute')
>>> A
{'a', 's', 'c', 'd'}
>>> B
{'e', 'i', 'n', 's', 'u', 't'}
>>> A - B # Set difference
{'a', 'c', 'd'}
>>> A | B # Set Union
{'a', 'c', 'e', 'd', 'i', 'n', 's', 'u', 't'}
>>> A & B # Set intersection
{'s'}
>>> A ^ B # Symmetric Difference
set(['a', 'd', 'c', 'e', 't', 'i', 'u', 'n'])
```

Sep-22

Programming

87

## Dictionaries

- Unordered set of **key:value** pairs,
- Keys have to be unique and immutable
- Key:value pairs enclosed inside curly braces {...}
- Empty dictionary is created by writing {}
- **Dictionaries are mutable**
  - add new key:value pairs,
  - change the pairing
  - delete a key (and associated value)

Sep-22

Programming

88



## Operations on Dictionaries

Operation	Meaning
len(d)	Number of key:value pairs in d
d.keys()	List containing the keys in d
d.values()	List containing the values in d
k in d	True if key k is in d
d[k]	Value associated with key k in d
d.get(k, v)	If k is present in d, then d[k] else v
d[k] = v	Map the value v to key k in d (replace d[k] if present)
del d[k]	Remove key k (and associated value) from d
for k in d	Iterate over the keys in d

Sep-22

Programming

89

## Operations on Dictionaries

```
>>> capital = {'India':'New Delhi', 'USA':'Washington DC', 'France':'Paris', 'Sri Lanka':'Colombo'}
>>> capital['India'] # Get an existing value
'New Delhi'
>>> capital['UK'] # Exception thrown for missing key

Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    capital['UK'] # Exception thrown for missing key
KeyError: 'UK'
>>> capital.get('UK', 'Unknown') # Use of default value with get
'Unknown'
>>> capital['UK']='London' # Add a new key:val pair
>>> capital['UK'] # Now it works
'London'
```

Sep-22

Programming

90

## Operations on Dictionaries

```
>>> capital.keys()
['Sri Lanka', 'India', 'UK', 'USA', 'France']
>>> capital.values()
['Colombo', 'New Delhi', 'London', 'Washington DC', 'Paris']
>>> len(capital)
5
>>> 'USA' in capital
True
>>> 'Russia' in capital
False
>>> del capital['USA']
>>> capital
{'Sri Lanka': 'Colombo', 'India': 'New Delhi', 'UK': 'London', 'France': 'Paris'}
```

Sep-22

Programming

91

## Operations on Dictionaries

```
>>> capital['Sri Lanka'] = 'Sri Jayawardenepura Kotte' # Wikipedia told me this!
>>> capital
{'Sri Lanka': 'Sri Jayawardenepura Kotte', 'India': 'New Delhi', 'UK': 'London', 'France': 'Paris'}

>>> countries = []
>>> for k in capital:
    countries.append(k)

# Remember: for ... in iterates over keys only

>>> countries.sort() # Sort values in a list
>>> countries
['France', 'India', 'Sri Lanka', 'UK']
```

Sep-22

Programming

92

## Dictionary Construction

- The **dict** constructor: builds dictionaries directly from *sequences of key-value pairs*

```
>>> airports=dict([('Mumbai', 'BOM'), ('Delhi', 'Del'), ('Chennai', 'MAA'), ('Kolkata', 'CCU')])
>>> airports
{'Kolkata': 'CCU', 'Chennai': 'MAA', 'Delhi': 'Del', 'Mumbai': 'BOM'}
```

Sep-22

Esc101, Strings

93

## Programming with Python

### File I/O

Sep-22

Programming

94

## File I/O

- Files are persistent storage
- Allow data to be stored beyond program lifetime
- The basic operations on files are
  - open, close, read, write
- Python treat files as sequence of lines
  - sequence operations work for the data read from files

Sep-22

Programming

95

## File I/O: **open** and **close**

**open**(filename, mode)

- While opening a file, you need to supply
  - The name of the file, including the path
  - The mode in which you want to open a file
  - Common modes are **r** (read), **w** (write), **a** (append)
- Mode is optional, defaults to **r**
- open**(..) returns a file object
- close**() on the file object closes the file
  - finishes any buffered operations

Sep-22

Programming

96



## File I/O: Example

```
>>> players = open('tennis_players', 'w')
>>>
>>> • Do some writing
>>> • How to do it?
>>> • see the next few slides
>>> players.close() # done with writing
```

Sep-22

Programming

97

## File I/O: read, write and append

- Reading from an open file returns the contents of the file
  - as **sequence** of lines in the program
- Writing to a file
  - **IMPORTANT:** If opened with mode **'w'**, **clears** the existing contents of the file
  - Use append mode (**'a'**) to preserve the contents
  - Writing happens at the end

Sep-22

Programming

98

## File I/O: Examples

```
>>> players = open('tennis_players', 'w')
>>> players.write('Roger Federar\n')
>>> players.write('Rafael Nadal\n')
>>> players.write('Andy Murray\n')
>>> players.write('Novak Djokovic\n')
>>> players.write('Leander Paes\n')
>>> players.close() # done with writing

>>> countries = open('tennis_countries', 'w')
>>> countries.write('Switzerland\n')
>>> countries.write('Spain\n')
>>> countries.write('Britain\n')
>>> countries.write('Serbia\n')
>>> countries.write('India\n')

>>> countries.close() # done with writing
```

Sep-22

Programming

99

## File I/O: Examples

```
>>> print(players)
<closed file 'tennis_players', mode 'w' at 0x031A48B8>
>>> print(countries)
<closed file 'tennis_countries', mode 'w' at 0x031A49C0>

>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
>>> n
<open file 'tennis_players', mode 'r' at 0x031A4910>
>>> c
<open file 'tennis_countries', mode 'r' at 0x031A4A70>
```

Sep-22

Programming

100

```
>>> pn = n.read() # read all players
>>> pn
'Roger Federar\nRafael Nadal\nAndy Murray\nNovak Djokovic\nLeander Paes\n'
>>> print(pn)
Roger Federar
Rafael Nadal
Andy Murray
Novak Djokovic
Leander Paes
>>> |
>>> n.close()
```

Note empty line due to '\n'

Sep-22

Programming

101

## File I/O: Examples

```
>>> n = open('tennis_players', 'r')
>>> c = open('tennis_countries', 'r')
>>> pn, pc = [], []
>>> for l in n:
    pn.append(l[:-1]) # ignore '\n'
>>> n.close()
>>> for l in c:
    pc.append(l[:-1])
>>> c.close()
>>> print(pn, '\n', pc)
['Roger Federar', 'Rafael Nadal', 'Andy Murray', 'Novak Djokovic', 'Leander Paes']
['Switzerland', 'Spain', 'Britain', 'Serbia', 'India']
```

Note the use of for ... in for sequence

Sep-22

Programming

102

## File I/O: Examples

```
>>> name_country = []
>>> for i in range(len(pn)):
    name_country.append((pn[i], pc[i]))
>>> print(name_country)
[('Roger Federar', 'Switzerland'), ('Rafael Nadal', 'Spain'), ('Andy Murray', 'Britain'), ('Novak Djokovic', 'Serbia'), ('Leander Paes', 'India')]
>>> n2c = dict(name_country)
>>> print(n2c)
{'Roger Federar': 'Switzerland', 'Andy Murray': 'Britain', 'Leander Paes': 'India', 'Novak Djokovic': 'Serbia', 'Rafael Nadal': 'Spain'}
>>> print(n2c['Leander Paes'])
India
```

Sep-22

Programming

103

## Programming using Python

## Modules and Packages

Welcome

Python Programming

104

## Modules

- As program gets longer, need to organize them for easier access and easier maintenance.
- Reuse same functions across programs without copying its definition into each program.
- Python allows putting definitions in a file
  - use them in a script or in an interactive instance of the interpreter
- Such a file is called a *module*
  - definitions from a module can be *imported* into other modules or into the *main* module

Sep-22

Programming

105

## Modules

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix `.py` appended.
- Within a module, the module's name is available in the global variable `__name__`.

Sep-22

Programming

106

## Modules Example



fib.py - C:\

fib.py - C:\Users\karkare\Google Drive\IITK\Courses\2016Python\Programs\fib.py (2.7.12)

File Edit Format Run Options Window Help

```
# Module for fibonacci numbers

def fib_rec(n):
    '''recursive fibonacci'''
    if (n <= 1):
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

Sep-22

Programming

107

## Modules Example

```
def fib_rec(n):
    '''recursive fibonacci'''
    if (n <= 1):
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)

def fib_iter(n):
    '''iterative fibonacci'''
    cur, nxt = 0, 1
    for k in range(n):
        cur, nxt = nxt, cur+nxt
    return cur

def fib_upto(n):
    '''given n, return list of fibonacci
    numbers <= n'''
    cur, nxt = 0, 1
    lst = []
    while (cur < n):
        lst.append(cur)
        cur, nxt = nxt, cur+nxt
    return lst
```

```
>>> import fib
>>> fib.fib_upto(5)
[0, 1, 1, 2, 3]
>>> fib.fib_rec(10)
55
>>> fib.fib_iter(20)
6765
>>> fib.__name__
'fib'
```

Within a module, the module's name is available as the value of the global variable

`__name__`.

Sep-22

Programming

108

## Importing Specific Functions

- To import specific functions from a module

```
>>> from fib import fib_upto
>>> fib_upto(6)
[0, 1, 1, 2, 3, 5]
>>> fib_iter(1)
```

```
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    fib_iter(1)
NameError: name 'fib_iter' is not defined
```

- This brings only the imported functions in the current symbol table
  - No need of `modulename`. (absence of `fib.` in the example)

Sep-22

Programming

109

## Importing ALL Functions

- To import *all* functions from a module, in the current symbol table

```
>>> from fib import *
>>> fib_upto(6)
[0, 1, 1, 2, 3, 5]
>>> fib_iter(8)
21
```

- This imports all names **except those beginning with an underscore (`_`)**.

Sep-22

Programming

110

## `__main__` in Modules

- When you run a module on the command line with `python fib.py <arguments>` the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`.
- By adding this code at the end of your module
 

```
if __name__ == "__main__":
    ... # Some code here
```

 you can make the file usable as a script as well as an importable module

Sep-22

Programming

111

## `__main__` in Modules

```
if __name__ == "__main__":
    import sys
    print (fib_iter(int(sys.argv[1])))
```

- This code parses the command line only if the module is executed as the “main” file:

```
$ python fib.py 10
55
```

- If the module is imported, the code is not run:

```
>>> import fib
>>>
```

Sep-22

Programming

112

## Package

- A Python package is a collection of Python modules.
- Another level of *organization*.
- *Packages* are a way of structuring Python's module namespace by using *dotted module names*.
  - The module name A.B designates a submodule named B in a package named A.
  - The use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Sep-22

Programming

113

## A sound Package

```

sound/
  __init__.py      Top-level package
                   Initialize the sound package
  formats/         Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/         Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/         Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

<https://docs.python.org/3/tutorial/modules.html>

Sep-22

Programming

114

## A sound Package

```

sound/
  __init__.py      Top-level package
                   Initialize the sound package
  formats/         Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/         Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/         Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

What are these files with funny names?

<https://docs.python.org/3/tutorial/modules.html>

Sep-22xs

Programming

115

## \_\_init\_\_.py

- The `__init__.py` files are required to make Python treat directories containing the file as packages.
- This prevents directories with a common name, such as `string`, unintentionally hiding valid modules that occur later on the module search path.
- `__init__.py` can just be an empty file
- It can also execute initialization code for the package

Sep-22

Programming

116

## Importing Modules from Packages

```

sound/
  __init__.py      Top-level package
                   Initialize the sound package
  formats/         Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/         Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/         Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

<https://docs.python.org/3/tutorial/modules.html>

Sep-22

Programming

117

## Importing Modules from Packages

```
import sound.effects.echo
```

- Loads the submodule `sound.effects.echo`
- It must be referenced with its full name:

```

sound.effects.echo.echofilter(
    input, output,
    delay=0.7, atten=4
)

```

Sep-22

Programming

118

## Importing Modules from Packages

```
from sound.effects import echo
```

- This also loads the submodule `echo`
- Makes it available without package prefix
- It can be used as:

```

echo.echofilter(
    input, output,
    delay=0.7, atten=4
)

```

Sep-22

Programming

119

## Importing Modules from Packages

```
from sound.effects.echo import echofilter
```

- This loads the submodule `echo`, but this makes its function `echofilter()` directly available.

```

echofilter(input, output,
           delay=0.7, atten=4)

```

Sep-22

Programming

120

## Popular Packages

- pandas, numpy, scipy, matplotlib, ...
- Provide a lot of useful functions

## Credit

Amey Karkare ,Dept. of CSE,IIT Kanpur  
<http://www.cse.iitk.ac.in/~karkare>