

# A Quick Tour of Python

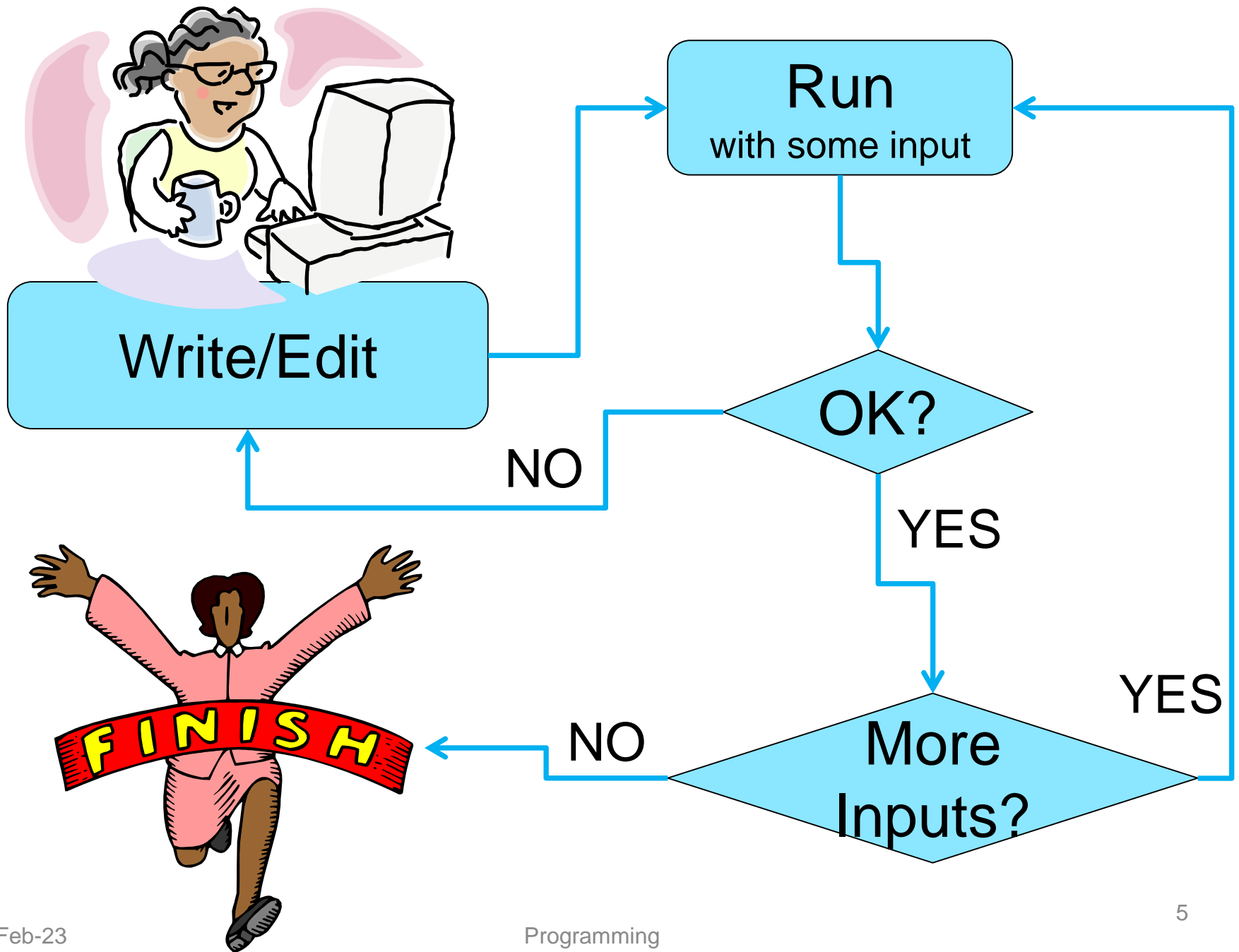
# Acknowledgements

- MS Office clip art, various websites and images
- The images/contents are used for teaching purpose and for fun. The copyright remains with the original creator. If you suspect a copyright violation, bring it to my notice and I will remove that image/content.

# About me

<http://www.cse.iitk.ac.in/~karkare>

# The Programming Cycle for Python



```
1 print ('Welcome')
2 print ('to Acads')
3
```

**User Program**

**Filename, preferred extension is **py****

**IN[1] :**  **Python Shell Prompt**

Welcome  
to Acads

**IN[2] :**

 **Outputs**

# Python Shell is Interactive

# 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`)



# 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?')
```

```
IN[2]:
```

```
IN[3]:
```

# 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**)

# 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'

# 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)

# Example of Types

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

```
Out[14]: int
```

# 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

# 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'
```

# 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)
```

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



# Operators

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

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

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

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

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

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

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

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

# Variables

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

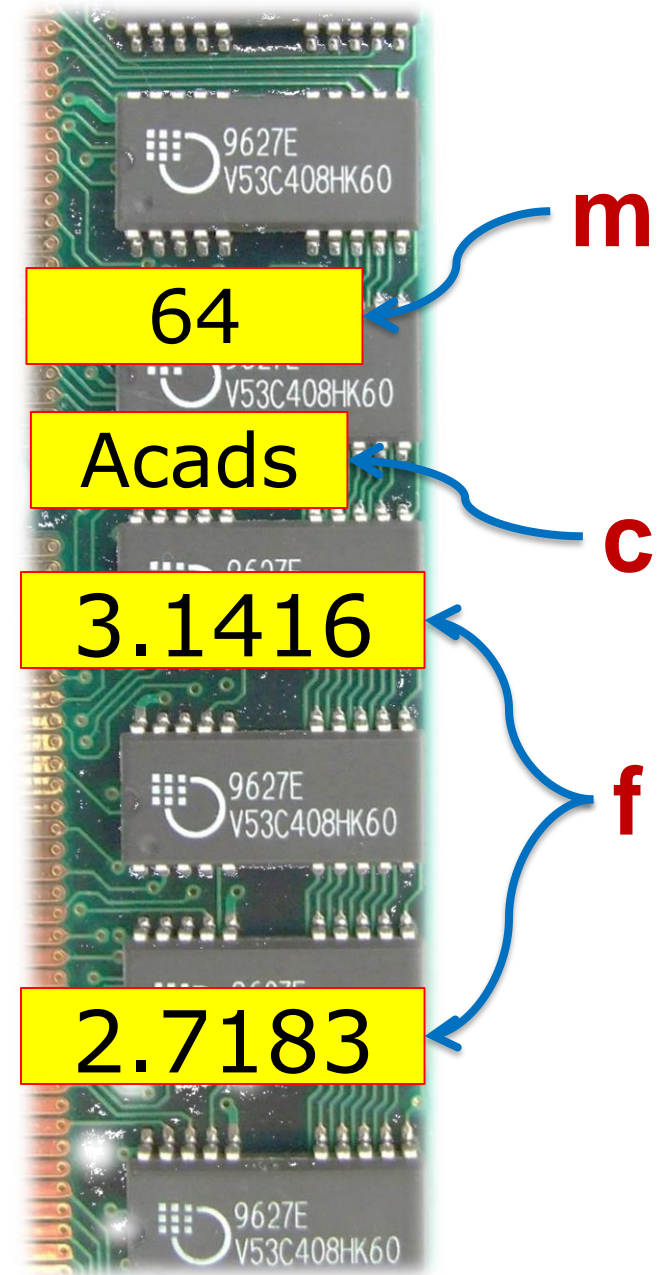
`m = 64;`

`c = 'Acads';`

`f = 3.1416;`

- Variables can change their bindings

`f = 2.7183;`



# 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.

# 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

# Programming using Python

## Operators and Expressions

# 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 4.25	In Python3
		9.1/2.0 is 4.55	Real div.
//	Integer Division	9//2 is 4	
%	Remainder	9%2 is 1	

# 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

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

# The % operator

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

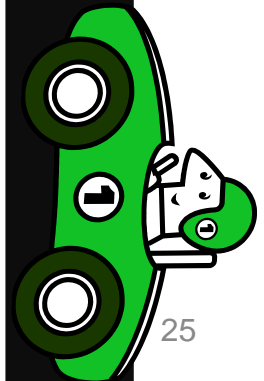
<b>9%4 is 1</b>	<b>(-1)%2 is 1</b>	<b>(-1)//(-2) is 0</b>
<b>9%4.5 is 0.0</b>	<b>1%(-2) is 1</b>	<b>1%0.6 is 0.4</b>

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



# 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.



# 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.

# 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 ('The min')`

**skipped**

Run the program

6

10

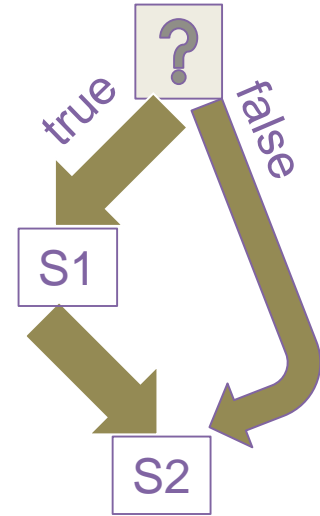
Output

**6**

# if statement (no else!)

- General form of the if statement

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

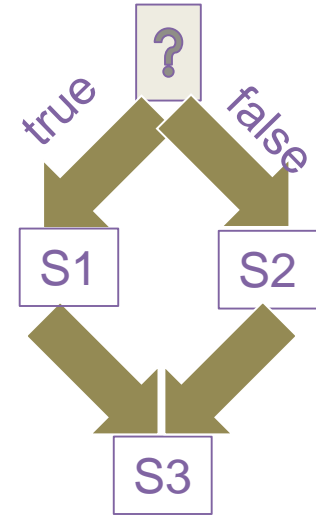


- Execution of if statement
  - First the expression is evaluated.
  - If it evaluates to a **true** value, then S1 is executed and then control moves to the S2.
  - If expression evaluates to **false**, then control moves to the S2 directly.

# 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!

# Nested if, if-else

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

# 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:
            ...
```

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

# Summary of if, if-else

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



# Class Quiz

- What is the value of expression:

$(5 < 2)$  and  $(3/0 > 1)$

a) Run time crash/error

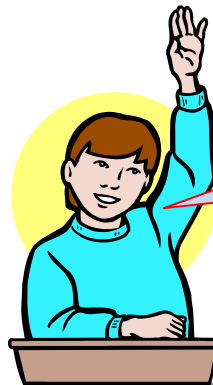


b) I don't know / I don't care



c) False

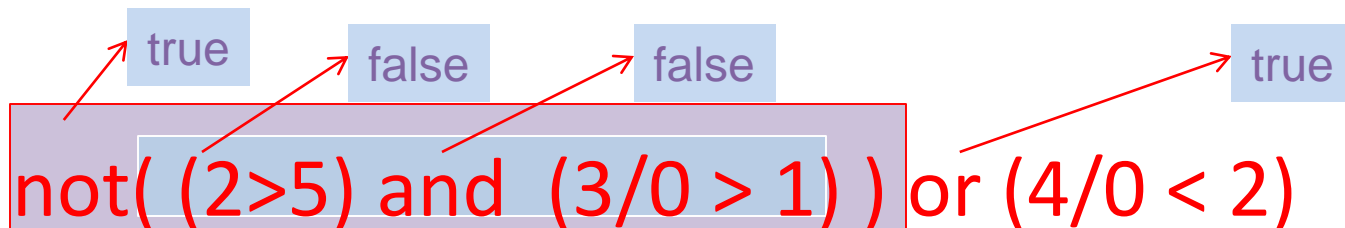
d) True



The correct answer is  
**False**

# 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



Evaluates to true

# 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.

# 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

*0.10000000000000000055511151231257827021181583404541015625*

- Approximation is similar to decimal approximation  $1/3 = 0.3333333333...$
- 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

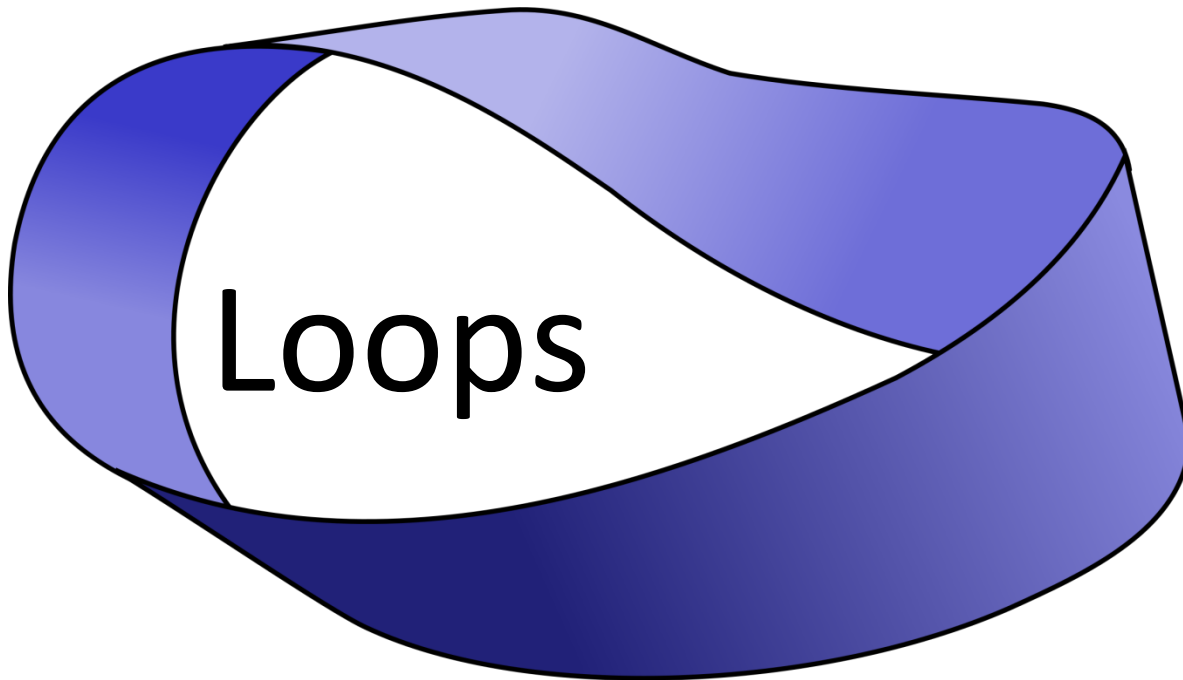
$x == y$

use

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

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

# Programming using Python





# 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

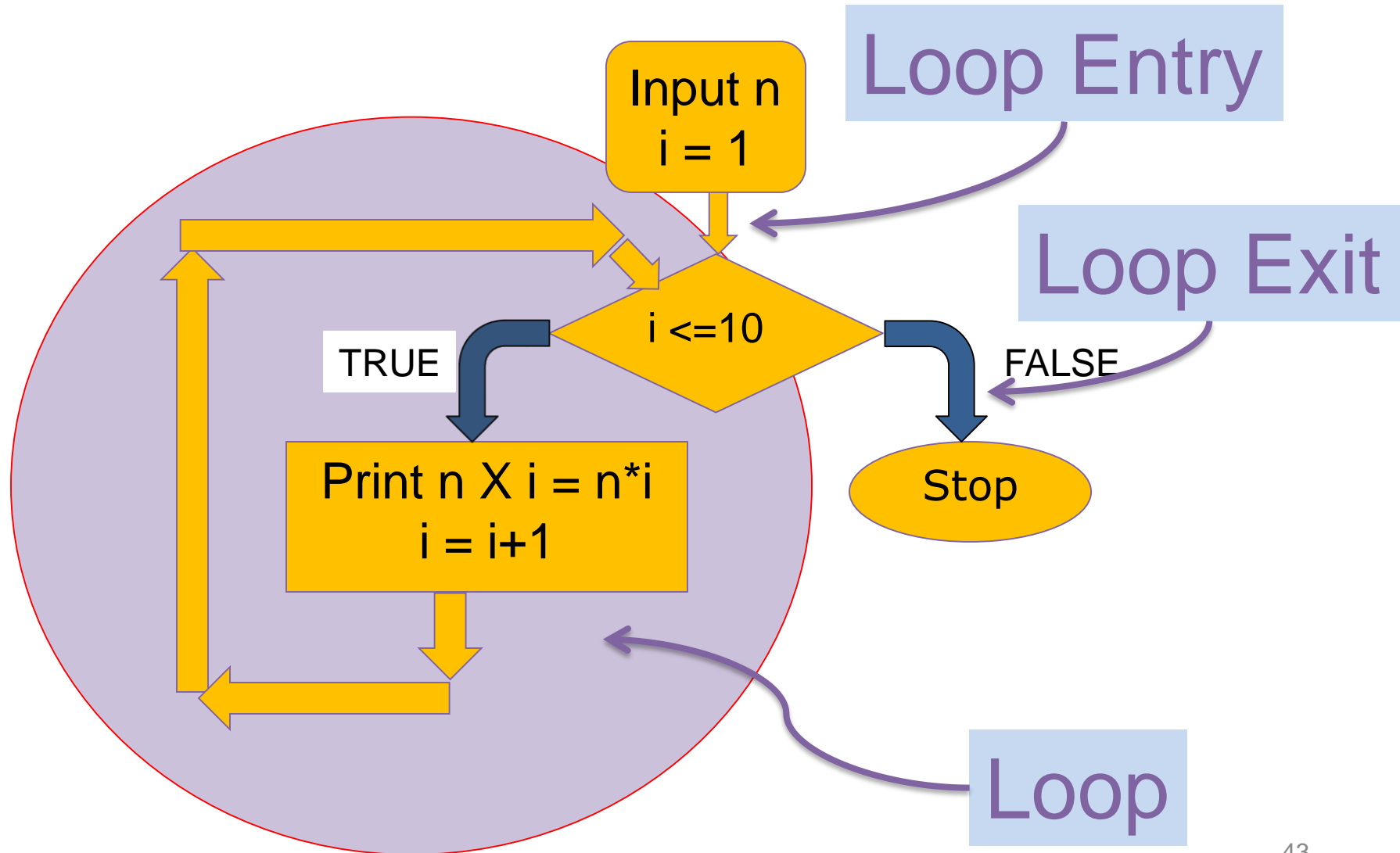
# Program...

```
n = int(input('Enter a number: '))
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)
....
```

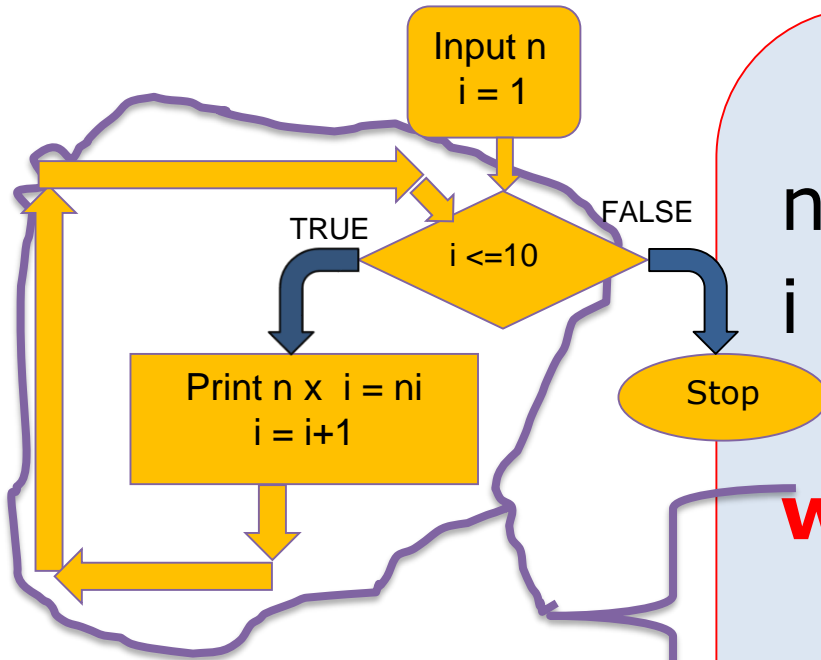


**Too much  
repetition!  
Can I avoid  
it?**

# Printing Multiplication Table



# Printing Multiplication Table



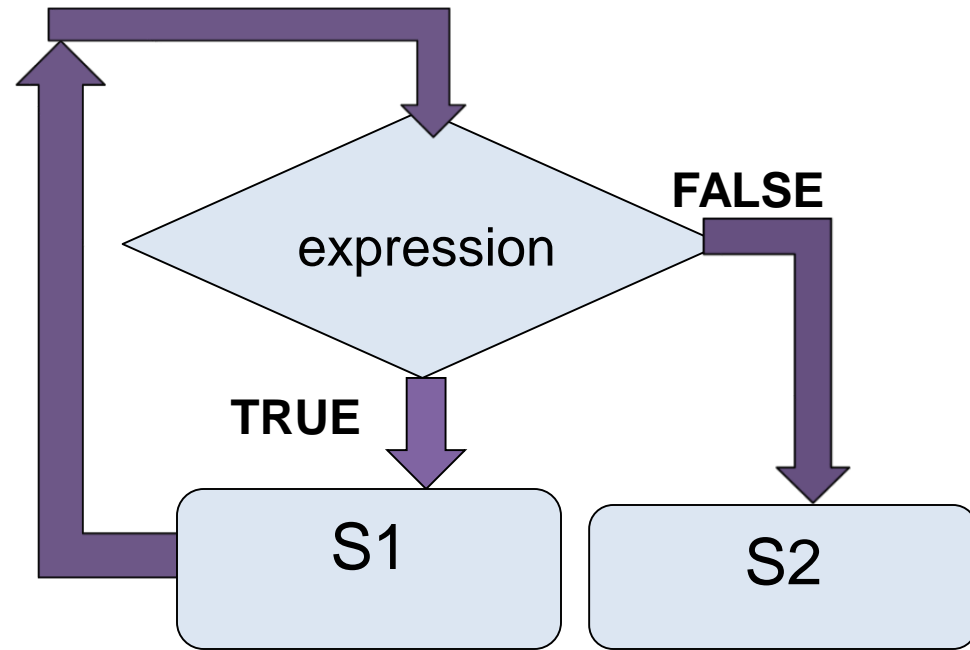
```
n = int(input('n=? '))
```

```
i = 1
```

```
while (i <= 10) :  
    print (n , 'X', i, '=', n*i)  
    i = i + 1  
print ('done')
```

# 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.

# 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)
```

# For loop in Python

- General form

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

# 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.



# 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
```

# 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= ' ')
```

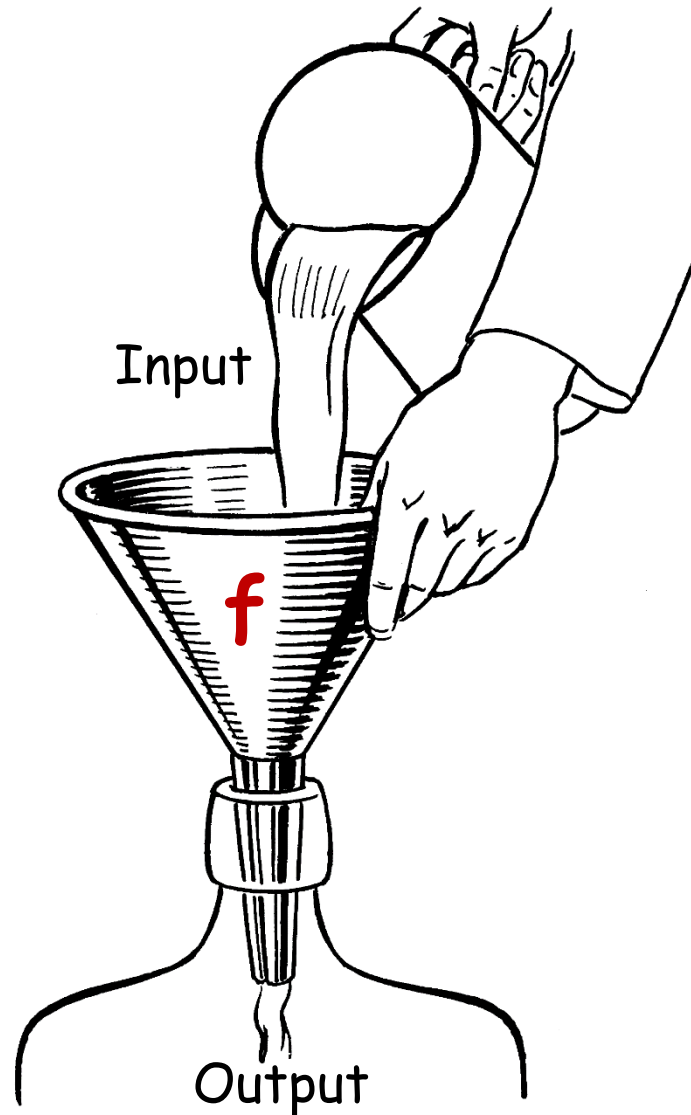
```
    i = i+1
```

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

# Programming using Python

**f**(unctions)

# Parts of a function



```
def max(a, b):
```

```
    "return maximum among a and b"
```

```
    if (a > b):  
        return a  
    else:  
        return b
```

keyword

2 arguments  
a and b  
(formal args)

Function Name

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

```
x = max(6, 4)
```

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

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

```
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
```

# 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
printName('Acads', 'Institute', False)	Acads Institute

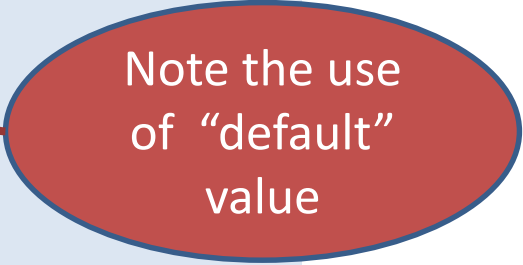
# Keyword Arguments

- 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



# 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

`printName('Acads', 'Institute')`

## Output

Acads Institute

# 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

# 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.

# 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
```

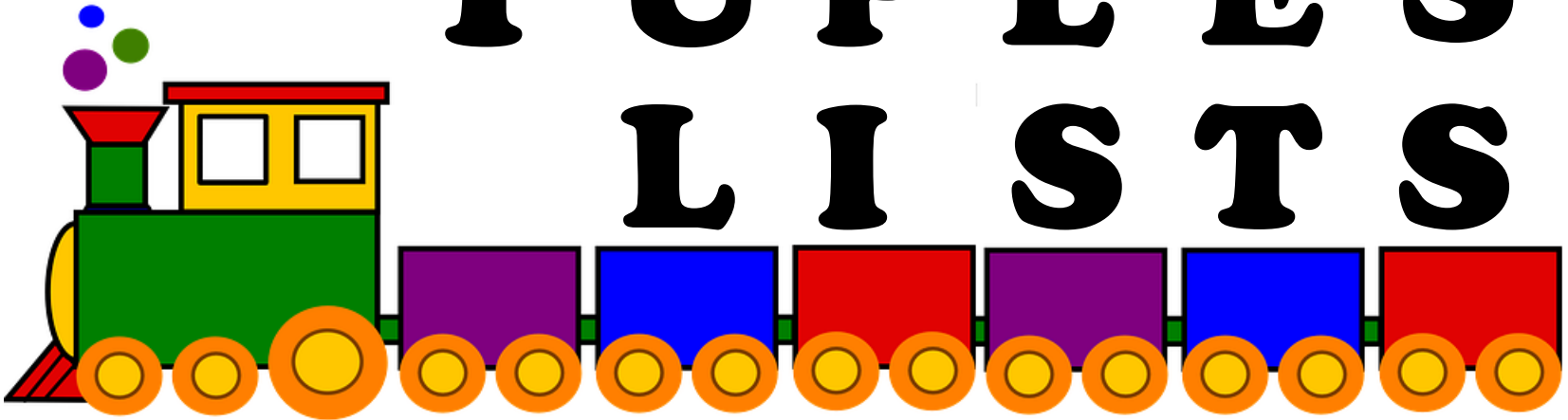
defines **PI** to be of float type with value 3.14. **PI** can be used across functions. Any change to **PI** in **update\_pi** will be visible to all due to the use of **global**.

# Programming with Python

**S T R I N G S**

**T U P L E S**

**L I S T S**



# 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

# Strings

```
>>> name='intro to python'  
>>> descr='acad\'s first course'
```

- More readable when **print** is used

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

# Length of a String

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

```
>>> name='intro to python'
>>> empty=''
>>> single='a'
```

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



# 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

# Concatenate and Repeat

```
>>> details = name + ', ' + descr
```

```
>>> details
```

```
"intro to python, acad's first course"
```

# Indexing

- Strings can be indexed
- First character has index 0

```
>>> name='Acads'
```

# Indexing

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

```
>>> name='Acads'
```

```
>>> name[-1]
```

```
's'
```

```
>>> name[-5]
```

```
'A'
```

```
>>> name[-2]
```

```
'd'
```

# Indexing

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

# 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`

# Slicing

```
>>> name='Acads'  
>>> name[0:3]
```

# 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	



# 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  $\geq$  end, empty slice is returned

Why?



# Tuples

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

```
>>> t = 'intro to python', 'amey karkare', 101
```

- Empty and Singleton Tuples

# Nested Tuples

- Tuples can be nested
- Note that **course** tuple is copied into **student**.
  - Changing **course** does not affect **student**

# 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
```

# More Operations on Tuples

- Tuples can be concatenated, repeated, indexed and sliced

```
>>> 2*course1  
('Python', 'Amey', 101, 'Python', 'Amey', 101)
```

# 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

# Unpacking Sequences

```
>>> student  
('Prasanna', 34, ('Python', 'Amey', 101))  
>>> name, roll, regdcourse=student  
>>> name
```

# 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'>
```



# Lists

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

# Lists

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

```
>>> [0] + fib # Concatenation
```

# More Operations on Lists

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

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

# 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

# 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.

# Summary of Sequences

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

# Programming with Python

## Sets and Dictionaries


# Sets

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



# Sets

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



**Create a set from  
a sequence**

# Set Operations

```
>>> A=set('acads')
>>> B=set('institute')
>>> A
{'a', 's', 'c', 'd'}
>>> B
{'e', 'i', 'n', 's', 'u', 't'}
```

# 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)

# Operations on Dictionaries

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

# Operations on Dictionaries

```
>>> capital = {'India':'New Delhi', 'USA':'Washington DC', 'France':'Paris', 'Sri Lanka':'Colombo'}
```

# Operations on Dictionaries

# Operations on Dictionaries

# 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'}
```