Disclaimers

All the code snippets, scripts, illustrations in this document '**Introduction to Python**' were prepared on Python IDLE 2.7.

If the same are run on newer versions of IDLE , they might result in different outputs.


python

Why is it called Python ?

- Creator of Python : **Guido van Rossum**
- Python : a snake that kills its prey through suffocation and can reach lengths of over 6 m/19 ft.
- He was also reading the published scripts from “**Monty Python’s Flying Circus**”, a BBC comedy series from the 1970s. so he decided to call the language Python.

What are Python’s strengths ?

- It’s Object-Oriented and Functional
- It’s Free
- It’s Portable
- It’s Powerful
- It’s Mixable
- It’s Relatively Easy to Use
- It’s Relatively Easy to Learn
- It’s Named After Monty Python

Who Uses Python Today?

- Google
- YouTube
- Dropbox
- Bit-Torrent
- Intel
- Cisco
- Hewlett-Packard
- Seagate
- Qualcomm
- IBM
- JPMorgan
- NASA
- & We

What Can I Do with Python?

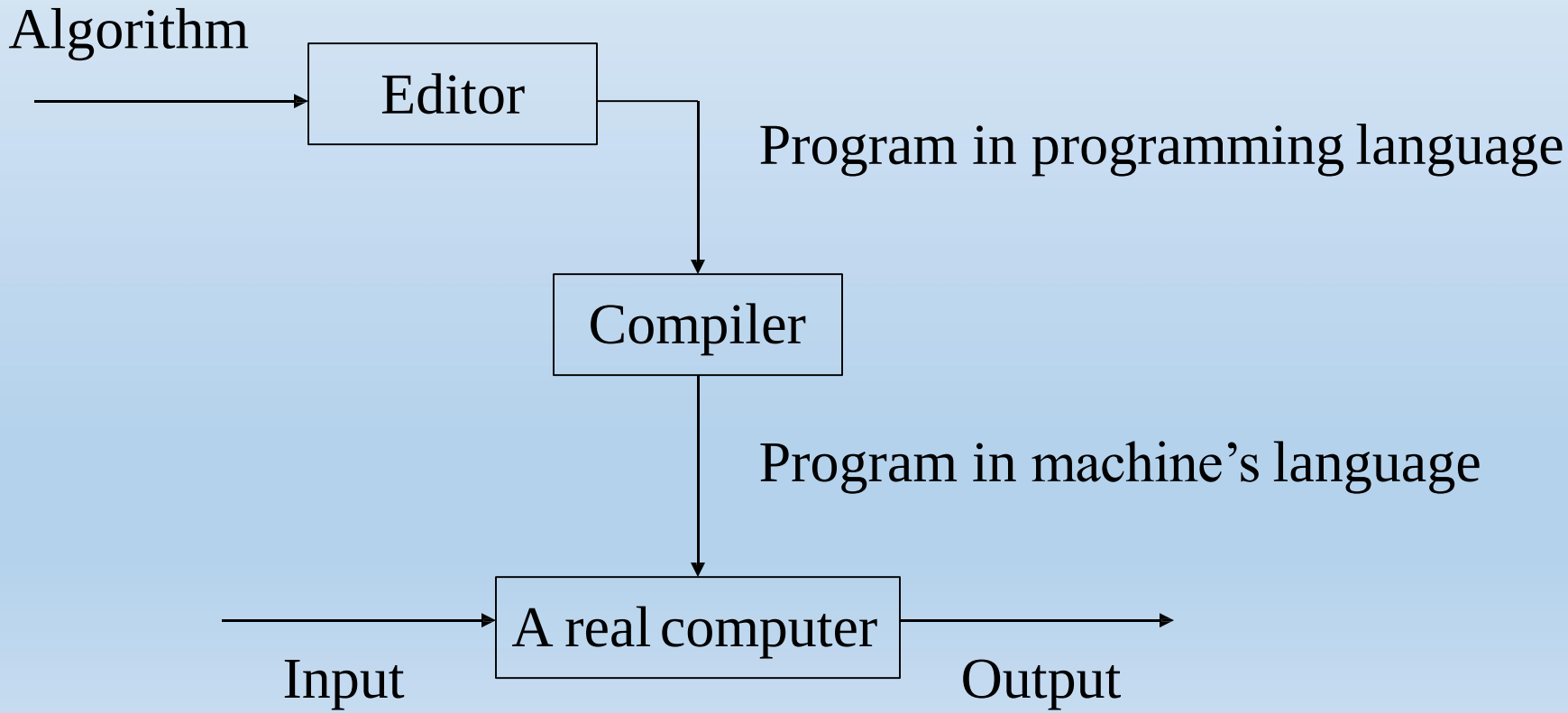
- Systems Programming
- GUIs
- Internet Scripting
- Component Integration
- Database Programming
- Rapid Prototyping
- Numeric and Scientific Programming
- And More: Gaming, Images, Data Mining, Robots, Excel...

Installation Process

- The current production versions are Python 3.4.1 and Python 2.7.8
- Download from below link
- https://www.python.org/downloads/release/python-341/ Or Search Google
- (Path Setting for Advance Users)

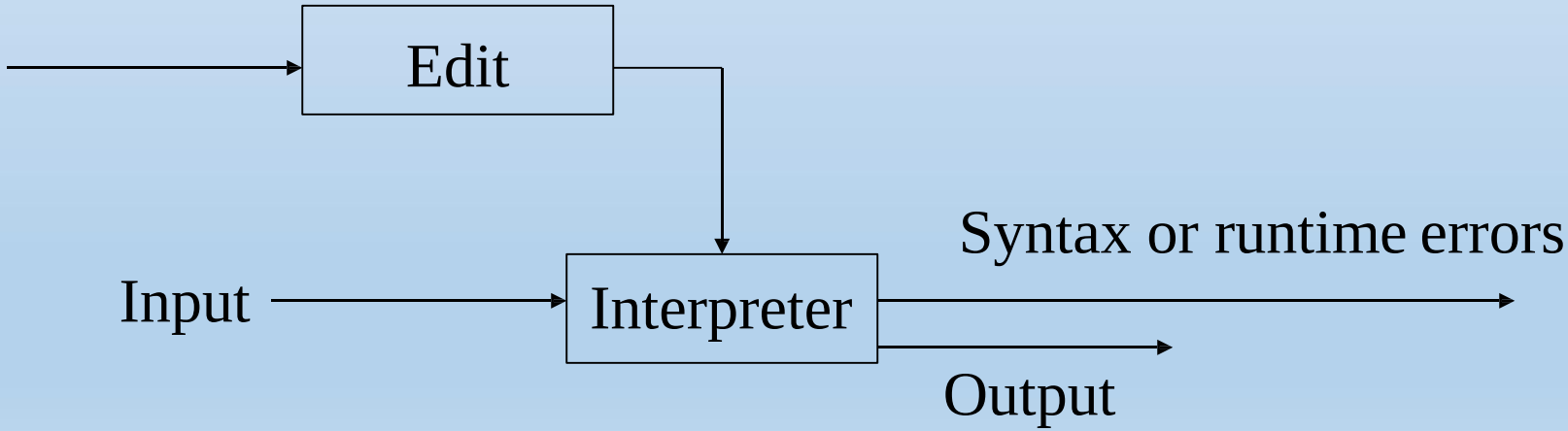
Compiler

- A *compiler* is a program that converts a program written in a programming language into a program in the native language, called *machine language*, of the machine that is to execute the program.



Interpreter

- An alternative to a compiler is a program called an *interpreter*. Rather than convert our program to the language of the computer, the interpreter takes our program one statement at a time and executes a corresponding set of machine instructions.



Three kinds of errors

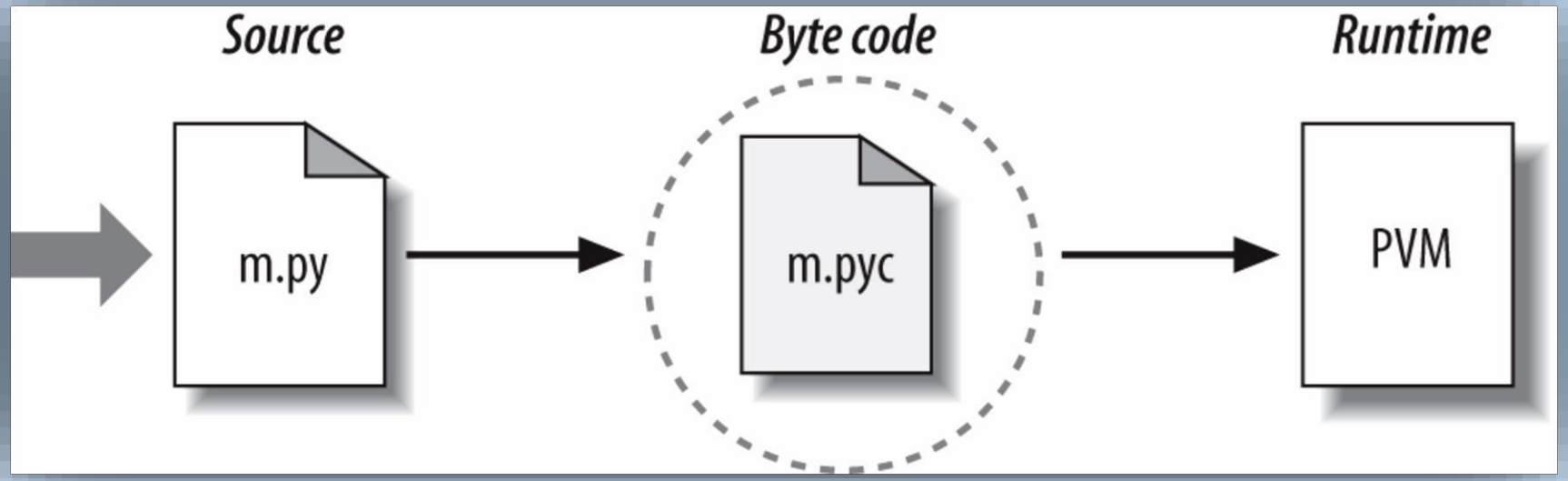
- *Syntax error* : Some statement in the program is not a legal statement in the language.
- *Runtime error* : An error occurs while the program is executing, causing the program to terminate (divide by zero, etc.)
- *Logic error* : The program executes to completion, but gives incorrect results.

Python’s runtime execution model

- Source code you type is translated to byte code.

  Which is then run by the Python Virtual Machine.
- Your code is automatically compiled

  But then it is interpreted.



Python IDLE

Integrated Development Environment

IDLE provides two modes in which to work:

- Interactive Mode
- Script Mode

Your First Program on Command Line

- Type the below given sentence & press enter :

```
print ("Hello World !!")
```

- Symbol ">>" waiting for your command expression
- Results of your expressions are displayed below the >>> input lines after you press the Enter key.
- Doesn't save your code in a file.
- Used for experiments & tests.
- The interactive prompt runs one statement at a time.

Your First Program on GUI

- Open File > New file (Ctrl + N) Type the above code line & save it & name it.
- Click on Run > Run Module or Press F5.

- Auto-indent and unindent for Python code in the editor (Backspace goes back one level)
- Word auto-completion while typing, invoked by a Tab press
- Balloon help pop ups for a function call when you type its opening "("

Comments In Python

Types Of Comments in Programming Languages:

1.Single Line Comments 2.Multi Line Comments

Comments in Python start with the hash character, #, and extend to the end of the physical line. Anything after the # is ignored by python.

is called Pound character

```
>> # print "This won't run."
>> print "This will run."
>> print "Hi # there.“
text = "# This is not a comment because it's inside quotes."
```

Elements of Program

These are all different, valid names

- X
- Celsius
- Spam
- spam
- spAm
- Spam_and_Eggs
- Spam_And_Eggs

Elements of Program

- Some identifiers are part of Python itself. These identifiers are known as reserved words. This means they are not available for you to use as a name for a variable, etc. in your program.
- and, del, for, is, raise, assert, elif, in, print, etc.

Expressions

- The fragments of code that produce or calculate new data values are called expressions.
- Literals are used to represent a specific value, e.g. 3.9, 1, 1.0
- Simple identifiers can also be expressions.

Python Keywords

and	elif	If	print
as	else	import	raise
assert	except	In	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

Elements of Program

```
print(3+4)                                  7
print(3, 4, 3+4)                            3 4 7
print()
print(3 + 4)                                7
print("The answer is ", 3+4)                 The answer is 7
```

Type Conversion Functions

float(<expr>) :

Convert expr to a floating point value

int(<expr>) :

Convert expr to an integer value

str(<expr>) :

Return a string representation of expr

eval(<string>) :

Evaluate string as an expression

Type Conversion

```
>>> float(22//5)
4.0
>>> int(4.5)
4
>>> int(3.9)
3
>>> round(3.9)
4
>>> round(3)
3
```

Assigning Input

- The purpose of an input statement is to get input from the user and store it into a variable.

<variable> = input(<prompt>)

<variable> = eval(input(<prompt>))

- First the prompt is printed
- The input part waits for the user to enter a value and press <enter>
- The expression that was entered is evaluated to turn it from a string of characters into a Python value (a number).
- The value is assigned to the variable.

Assigning Input

```
a = input ("Enter your name : ")
b = int (input ("Enter number : "))
c = float (input("Enter number : "))
d = eval (input ("Enter number : "))

Enter your name : Rajdeep
Enter number : 5
Enter number : 4
Enter number : 8
>>> a
'Rajdeep'
>>> b
5
>>> c
4.0
>>> d
8
>>>
```

Python’s Core Data Types

Python has five standard data types:

- Numbers
- String
- List
- Tuple
- Dictionary

Numbers

- There are three distinct numeric types:
- integers, floating point numbers, and complex numbers.
- In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using double in C;
- The integer numbers (e.g. 2, 4, 20) have type int ,
- The ones with a fractional part (e.g. 5.0, 1.6) have type float.
- Complex numbers, uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

Numbers

Python has a special function to tell us the data type of any value.

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = 32
>>> type(myInt)
<class 'int'>
```

Operation	Result
x + y	sum of *x* and *y*
x - y	difference of *x* and *y*
x * y	product of *x* and *y*
x / y	quotient of *x* and *y*
x // y	floored quotient of *x* and *y*
x % y	remainder of x / y
-x	*x* negated
+x	*x* unchanged
x ** y	*x* to the power *y*

abs(x)	absolute value or magnitude of *x*
int(x)	*x* converted to integer
float(x)	*x* converted to floating point
complex(re, im)	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
divmod(x, y)	the pair (x // y, x % y)
pow(x, y)	*x* to the power *y*
math.trunc(x)	*x* truncated to Integral
round(x, n)	*x* rounded to n digits, rounding half to even. If n is omitted, it defaults to 0.
math.floor(x)	the greatest integral float <= *x*
math.ceil(x)	the least integral float >= *x*

Operations on Numbers

The integer numbers (e.g. 2, 4, 20) have type int , the ones with a fractional part (e.g. 5.0, 1.6) have type float.

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5
1.6
```

Operations on Numbers

- >>> 3.0+4.0
- 7.0
- >>> 3+4
- 7
- >>> 3.0*4.0
- 12.0
- >>> 3*4
- 12

- >>> 10.0/3.0
- 3.3333333333333335
- >>> 10/3
- 3.3333333333333335
- >>> 10 // 3
- 3
- >>> 10.0 // 3.0
- 3.0

Operations on Numbers

```
>>> 17 / 3     # classic division returns a float          5.666666666666667
>>> 17 // 3    #  floor division  discards  the fractional  part                  5
>>> 17 % 3     # the % operator returns the remainder of    the  division    2
>>>  5 * 3  +  2        # result * divisor + remainder                          17
```

Division (/) always returns a float.

To do floor division and get an integer result (discarding any fractional result) you can use the // operator

To calculate the remainder you can use %

Operations on Numbers

With Python, it is possible to use the ** operator to calculate powers.

- >>> 5 ** 2 *# 5 squared* *25*
- >>> 2 ** 7 *# 2 to the power of 7* *128*

*# Python defines pow(0, 0) and 0 ** 0 to be 1, as is common for programming languages*

- >>> 5+4J + 4+5J *#Add Complex Nos.* *9+9j*

Using Math Library

- To use a library, we need to make sure this line is in our program: import math
- Importing a library makes whatever functions are defined within it available to the program.
- To access the sqrt library routine, we need to access it as math.sqrt(x).
- Using this dot notation tells Python to use the sqrt function found in the math library module.

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Assignment of Variables

- The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

- >>> width = 20
- >>> height = 5 * 9
- >>> width * height
- 900

Undefined Declaration gives an Error

- If a variable is not “defined” (assigned a value), trying to use it will give you an error:

- >>> n *# try to access an undefined variable*
- Traceback (most recent call last):
- File "<stdin>", line 1, in <module>
- NameError: name 'n' is not defined

Int + Float = Float

- There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

- >>> 3 * 3.75 / 1.5
- 7.5
- >>> 7.0 / 2
- 3.5

_ is value of last printed expression

- In interactive mode, the last printed expression is assigned to the variable _.
- This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:
- >>> tax = 12.5 / 100
- >>> price = 100.50
- >>> price * tax
- 12.5625
- >>> price + _
- 113.0625

Short Hand Operators

These are short-hand notation for a common operation in programming

x += n	<-- stands for -->	x = x + n
x -= n	<-- stands for -->	x = x – n
x *= n	<-- stands for -->	x = x * n
x /= n	<-- stands for -->	x = x / n

Most commonly, this is used to increment or decrement a variable, like so:

a += 1	<-- increment variable a
b -= 1	<-- decrement variable b

Strings

- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').
- Python strings cannot be changed — they are immutable.
- Unlike other languages, special characters such as \n have the same meaning with both single ('...') and double ("...") quotes.
- The only difference between the two is that within single quotes you don’t need to escape " (but you have to escape \') and vice versa

```
>>> 'spam eggs'                 # single quotes
'spam eggs'
>>> 'doesn\'t'                  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"                   # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Escape Sequence \n

```
>>> s = 'First line\nSecond line'                    # \n means newline
>>> s                                  # without print(), \n is included in the output
'First line\nSecond line‘

>>> print(s)                                  # with print(), \n produces a new line
First line
Second line
```

r Before Quotes

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

```
>>> print('some\name')                    # here \n means newline!
some
ame
>>> print(r'some\name')                   # note the r before the quote
some\name
```

String Formatting

- Similar to C’s printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.
- >>> "One, %d, three" % 2
- 'One, 2, three'
- >>> "%d, two, %s" % (1,3)
- '1, two, 3'
- >>> "%s two %s" % (1, 'three')
- '1 two three'

Multiple lines

Using triple-quotes: """...""" or '''...'''.

 # \ after or before “”” remove end of the line

```
print("""
Hello
Python
Programmers
""")
```

Output :

```
Hello
Python
Programmers
```

Breaking Long Strings

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
            'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

String Concatenation

Strings can be concatenated (glued together) with the + operator :

```
>>> ‘python' + ‘programmers'
‘pythonprogrammers'
>>> code = ‘py’
>>> code + 'thon'
'Python‘
>>> 'Py' 'thon'
'Python‘
```

Repetition of Strings

Strings can be repeated with * :

```
>>> 3 * ‘code’                                    # 3 times ‘code’
codecodecode
```

In Python built-in function len() returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Operations on String

- We can access the individual characters in a string through *indexing*.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.
- In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.
- We can index from the right side using negative indexes.

String Indexing

```
>>> word = 'Python‘
>>> word[0]                                  # character in position 0
'P'
>>> word[3]                                  # character in position 5
‘h‘
>>> word[5]                                  # character in position 0
‘n'
>>> word[6]                          # IndexError : string index out of range
'n'
```

String Indexing(Contd.)

```
>>> word[-1]                                    # last character
'n'
>>> word[-2]                                    # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

String Slicing

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.
- Slicing:
  <string>[<start>:<end>]
- start and end should both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.

String Slicing

In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain substring:

The start is always included, and the end always excluded.

```
>>>   word[0:2]# characters from position 0 (included) to 2 (excluded)
'Py'
>>>   word[2:5]# characters from position 2 (included) to 5 (excluded)
'tho'
```

String Slicing (Contd.)

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[2:]
‘thon'
>>> word[4:]
‘on'
>>> word[:2]
'Py'
>>> word[:4]
'Pyth'
```

ASCII Values

The ord function returns the numeric (ordinal) code of a single character.

The chr function converts a numeric code to the corresponding character.

```
>>> ord("A")
65
>>> ord("a")
97
>>> chr(97)
'a'
>>> chr(65)
'A'
```

String Functions

- One of these methods is split. This will split a string into substrings based on spaces.
- >>> "Hello string methods!".split()
- ['Hello', 'string', 'methods!']
- Split can be used on characters other than space, by supplying the character as a parameter.
- >>> "32,24,25,57".split(",")
- ['32', '24', '25', '57']

String Functions

s.lower() – Copy of s in all lowercase letters

s.lstrip() – Copy of s with leading whitespace removed

s.replace(oldsub, newsub) – Replace occurrences of oldsub in s with newsub

s.rfind(sub) – Like find, but returns the right-most position

s.rstrip() – Copy of s with trailing whitespace removed

s.split() – Split s into a list of substrings

s.upper() – Copy of s; all characters converted to uppercase

String Functions

s.capitalize() – Copy of s with only the first character capitalized

s.title() – Copy of s; first character of each word capitalized

s.count(sub) – Count the number of occurrences of sub in s

s.find(sub) – Find the first position where sub occurs in s

s.join(list) – Concatenate list of strings into one large string using s as separator.

Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

i.e.

```
>>> squares = [1, 4, 9, 16, 25]
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> a = ['spam', 'eggs', 100, 1234]
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are mutable
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
```

Lists Indexing

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence type), lists can be indexed and sliced:

```
>>> squares[0]                    # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]                  # slicing returns a new list
[9, 16, 25]
```

Lists Slicing & Concatenating

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2] 100
```

```
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs',
100, 'spam', 'eggs', 100, 'Boo!']
```

Lists are Mutable

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

List Size is Changeable

```
>>> a[0:2] = [1, 12]
# Replaced some items
>>> a
[1, 12, 123, 1234]
>>> a[0:2] = []
# Remove some
>>> a
[123, 1234]
>>> a[1:1] = [‘C', ‘C++']
# Insert some
>>> a
[123, ‘C', ‘C++', 1234]
>>> a[:0] = a
# Insert (a copy of) itself at the beginning
>>> a
[123, ‘C', 'C++', 1234, 123, ‘C', 'C++', 1234]
>>> a[:] = []
# Clear the list: replace all items with an empty list
>>> a
[]
```

Lists Content Modification

x[i] = a reassigns the ith element to the value a

Since x and y point to the same list object, both are changed

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
```

Nested Lists

It is possible to nest lists (create lists containing other lists), for example:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

List Functions

myList.append(x)	Add x to end of myList
myList.sort()	Sort myList in ascending order
myList.reverse()	Reverse myList
myList.index(s)	Returns position of first x
myList.insert(i,x)	Insert x at position i
myList.count(x)	Returns count of x
myList.remove(x)	Deletes first occurrence of x
myList.pop(i)	Deletes and return ith element
x in myList	Membership check (sequences)

The ‘in’ Operator

Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

Be careful: the ‘in’ keyword is also used in the syntax of other unrelated Python constructions: “for loops” and “list comprehensions.”

The range() Function

- What does range(n) return?
  0, 1, 2, 3, …, n-1
- range has another optional parameter, range(start, n) returns
  start, start + 1, …, n-1
- But wait! There's more!
  range(start, n, step)
  start, start+step, …, n-1
- list(<sequence>) to make a list

The range() Function

Let’s try some examples!

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
```

Decision Structure

- Control structures allow us to alter this sequential program flow.
- In upcoming slides, we'll learn about decision structures, which are statements that allow a program to execute different sequences of instructions for different cases, allowing the program to "choose" an appropriate course of action.
- The Python if statement is used to implement the decision.

if <condition>:
    <body>

If Statement

- The body is a sequence of one or more statements indented under the if heading.
- The condition in the heading is evaluated.
- If the condition is true, the sequence of statements in the body is executed, and then control passes to the next statement in the program.
- If the condition is false, the statements in the body are skipped, and control passes to the next statement in the program.
- This is a one-way or simple decision.

If-else Statement

- In Python, a two-way decision can be implemented by attaching an `else` clause onto an `if` clause.
- This is called an `if-else` statement:

```
if <condition>:
    <statements>
else:
    <statements>
```

Two way Decision

- When Python first encounters this structure, it first evaluates the condition. If the condition is true, the statements under the if are executed.
- If the condition is false, the statements under the else are executed.
- In either case, the statements following the if-else are executed after either set of statements are executed

Multi Way Decisions

```
if <condition1>:
   <case1 statements>
elif <condition2>:
   <case2 statements>
elif <condition3>:
   <case3 statements>
else:
   <default statements>
```

Multi Way Decisions

- Python evaluates each condition in turn looking for the first one that is true. If a true condition is found, the statements indented under that condition are executed, and control passes to the next statement after the entire `if-elif-else`.
- If none are true, the statements under `else` are performed.
- The `else` is optional. If there is no `else`, it's possible no indented block would be executed.

Conditional Statement Example

```
if x1 >= x2:
    if x1 >= x3:
        max = x1
    else:
        max = x3
else:
    if x2 >= x3:
        max = x2
    else
        max = x3
```

```
if x1 >= x2:
    if x1 >= x3:
        max = x1
    else:
        max = x3
elif x2 >= x3:
        max = x2
else
        max = x3
```

While Loop

- `while <condition>:`
  `    <body>`
- `condition` is a Boolean expression, just like in `if` statements. The body is a sequence of one or more statements.
- Semantically, the body of the loop executes repeatedly as long as the condition remains true. When the condition is false, the loop terminates.

For Loop

- The `for` statement allows us to iterate through a sequence of values.
- `for <var> in <sequence>:`
  `<body>`
- The loop index variable `var` takes on each successive value in the sequence, and the statements in the body of the loop are executed once for each value.

Looping Example

- Here's an example of a while loop that counts from 0 to 10:

```
i = 0
while i < 11:
    print(i)
    i = i + 1
```

- The code has the same output as this for loop:

```
for i in range(11):
    print(i)
```

- The while loop requires us to manage the loop variable i by initializing it to 0 before the loop and incrementing it at the bottom of the body.
- In the for loop this is handled automatically.

Do-while loop

- When the condition test comes after the body of the loop it's called a *do while loop*.
- A *do while* loop always executes the body of the code at least once.
- Python doesn't have a built-in statement to do this, but we can do it with a slightly modified `while` loop.

Function

- A function is like a *subprogram*, a small program inside of a program.
- The basic idea – we write a sequence of statements and then give that sequence a name. We can then execute this sequence at any time by referring to the name.
- The part of the program that creates a function is called a *function definition*.
- When the function is used in a program, we say the definition is *called* or *invoked*.

Function

- A function definition looks like this:
  def <name>(<formal-parameters>):
      <body>
- The name of the function must be an identifier
- Formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside of the function body.

Function Example

```
def fun():
    print("Happy birthday to you!" )
    print("Happy birthday to you!" )
    print("Happy birthday, dear Fred...")
    print("Happy birthday to you!")
```

- Gives us this…

```
>>> fun()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

Function Call

- A function is called by using its name followed by a list of actual parameters or arguments.
  <name>(<actual-parameters>)

When Python comes to a function call, it initiates a four-step process

- The calling program suspends execution at the point of the call.
- The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
- The body of the function is executed.
- Control returns to the point just after where the function was called.

The Return Values

- This function returns the square of a number:

```
def square(x):
    return x*x
```

- When Python encounters `return`, it exits the function and returns control to the point where the function was called.
- In addition, the value(s) provided in the `return` statement are sent back to the caller as an expression result.

The Return Values

- Sometimes a function needs to return more than one value.
- To do this, simply list more than one expression in the `return` statement.
- all Python functions return a value, whether they contain a `return` statement or not. Functions without a `return` hand back a special object, denoted `None`.

```
def sumDiff(x, y):
    sum = x + y
    diff = x – y
    return sum, diff
```

Searching

- *Searching* is the process of looking for a particular value in a collection.
- Types of Searching :

1. Linear Search

2. Binary Search

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

Linear Search

- This strategy is called a *linear search*, where you search through the list of items one by one until the target value is found.
- ```
  def search(x, nums):
      for i in range(len(nums)):
          if nums[i] == x: # item found, return the index value
              return i
      return  -1           # loop finished, item was not in list
  ```
- This algorithm wasn't hard to develop, and works well for modest-sized lists.
  ```

Binary Search

- In this search the algorithm is a loop that looks at the middle element of the range, comparing it to the value `x`.
- If `x` is smaller than the middle item, `high` is moved so that the search is confined to the lower half.
- If `x` is larger than the middle item, `low` is moved to narrow the search to the upper half.
- The loop terminates when either
  - `x` is found
  - There are no more places to look (`low > high`)

Binary Search

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:          # There is still a range to search
        mid = (low + high)//2 # Position of middle item
        item = nums[mid]
        if x == item:           # Found it! Return the index
            return mid
        elif x < item:          # x is in lower half of range
            high = mid - 1      #  move top marker down
        else:                   # x is in upper half of range
            low = mid + 1       #  move bottom marker up
    return -1                   # No range left to search,
                                # x is not there
```

String Reversal

- ```
  def reverse(s):
      if s == "":
          return s
      else:
          return reverse(s[1:]) + s[0]
  ```
- ```
  >>> reverse("Hello")
  'olleH‘
  ```
- ```
  >>> “Hello”[::-1]
  'olleH‘
  ```
  ```

Tower of Hanoi

```
def TOI(n, source, dest, temp):
    if n == 1:
        print("Move disk from", source, "to", dest+".")
    else:
        TOI(n-1, source, temp, dest)
        TOI(1, source, dest, temp)
        TOI(n-1, temp, dest, source)
```

Thanks