PYTHON PROGRAMMING

(18IS5DEPYP)

UNIT -1

Mrs. Bhavani K

Assistant Professor

Dept. of ISE, DSCE

Unit 1

- **DATA TYPES**
  - Identifiers and Keywords
  - Integral Types
    - Integers
    - Booleans
  - Floating-Point Types
    - Floating-Point Numbers
    - Complex Numbers
    - Decimal Numbers
  - Strings
    - Comparing Strings
    - Slicing and Striding Strings
    - String Operators and Methods .
    - String Formatting with the str.format()Method
    - Character Encodings
  - Examples
    - quadratic.py
    - csv2html.py

DSCE
Dept. of Information Science & Engg

DATA TYPES

- Data type characterizes type of data present inside a variable.
  - Compared to other languages, data type of a variable in python need not be declared.

- Python has following standard Data Types:
  - Numeric
    - Integer
    - Float
    - Complex
  - Boolean
  - Sequence Type
    - String
    - List
    - Tuple
  - Set
  - Dictionary

>>> x =10 **>>> type(x)** **<class 'int'>**	**>>> x = 2.5** **>>> type(x)** **<class 'float'>**

Note: Python has an in-built function type() to ascertain the data type of a certain value.
Example: type(1234) in Python shell will return <class 'int'>

Identifiers and Keywords

DSCE
Dept. of Information Science & Engg

Identifiers and Keywords

- Python identifiers are user-defined names. They are used to specify the names of variables, functions, class, module, etc.

- Rules that must be followed to create a python identifier are:
  - Reserved keywords can't be used as an identifier name
  - Identifier can contain
    - letters in a small case (a-z)
    - upper case (A-Z)
    - digits (0-9) and
    - underscore (_)
  - Identifier name can't start with a digit
  - Identifier can't include only digits
  - Identifier name can start with an underscore
  - There is no limit on the length of the identifier name
  - Python identifier names are case sensitive

Identifier	Validity
My-name	invalid
2two	invalid
Y_name	Valid

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

Identifiers contd...

- **Underscore(_)** is a distinctive character in Python.
  - It has some special meaning in different conditions.
  - **Underscore(_)** can be used as a variable in looping
    - **Example:**
```
for _ in range(5):
    print(_)
```

Output:
0
1
2
3
4

- **Note:**
  - If identifier starts with _ symbol, it indicates that it is private
  - If identifier starts with __(two under score symbols) , it indicates that it is strongly private identifier.
  - If the identifier starts and ends with two underscore symbols then the identifier is language defined special name, which is also known as magic method.

Identifiers contd...

- Python has a built-in function called dir() that returns a list of an object's attributes.
  - If it is called with no arguments it returns the list of Python's built-in attributes.
  - Example:
    - >>> dir()

      ['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']

Integral Types

Integral (Numeric) Types

- Python provides two built-in integral types, int and bool.
  - When used in Boolean expressions,
    - 0 and False are False, and
    - any other integer and True are True.
  - When used in numerical expressions True evaluates to 1 and False to 0.
  - Example: i += True => i += 1.

Integer (Int)

- It is used to signify integral (whole) values. Numbers without decimal point are known as 'int' like 10, 20, 100 or 1000000 etc.
- The size of an integer is limited only by the machine's memory
- In 'int' we can represent values in 4 ways:
  - Decimal form
  - Binary form
  - Octal form
  - Hexa decimal form

Integer

Decimal form: (Base 10)
In decimal form, permissible digits are 0-9.
For example, a = 1111; by default Python is going to consider as decimal form
>>> a=1111
>>> print(a)
1111
	Binary form: (Base 2)
	In binary form, permissible digits are 0 and 1 only.
	Ex: a = 0b1111 or a = 0B1111
	>>> a = 0b1111
	>>> print(a)
	15

Octal form: (Base 8)
In octal form, permissible digits are 0 to 7.
Ex: a = 0o555 or a = 0O555
>>> a = 0o555
>>> print(a)
365
	Hexa decimal form: (Base-16)
	In hexa-decimal form, allowed digits are 0 to 9, a to f or A to F
	Ex: a = 0xabc or a = 0Xabc
	>>> a=0xFFF
	>>> print(a)
	4095

Numeric Operators and Functions

Syntax	Description
`x + y`	Adds number `x` and number `y`
`x - y`	Subtracts `y` from `x`
`x * y`	Multiplies `x` by `y`
`x / y`	Divides `x` by `y`; always produces a `float` (or a `complex` if `x` or `y` is `complex`)
`x // y`	Divides `x` by `y`; truncates any fractional part so always produces an `int` result; see also the `round()` function
`x % y`	Produces the modulus (remainder) of dividing `x` by `y`
`x ** y`	Raises `x` to the power of `y`; see also the `pow()` functions
`-x`	Negates `x`; changes `x`'s sign if nonzero, does nothing if zero
`+x`	Does nothing; is sometimes used to clarify code
`abs(x)`	Returns the absolute value of `x`
`divmod(x, y)`	Returns the quotient and remainder of dividing `x` by `y` as a tuple of two `ints`
`pow(x, y)`	Raises `x` to the power of `y`; the same as the `**` operator
`pow(x, y, z)`	A faster alternative to `(x ** y) % z`
`round(x, n)`	Returns `x` rounded to *n* integral digits if *n* is a negative `int` or returns `x` rounded to *n* decimal places if *n* is a positive `int`; the returned value has the same type as `x`; see the text

Integer

- When an object is created using its data type there are three possible use cases.
  - **1) a data type called with no arguments.**
    - An object with a default value is created.
      - example, x = int() creates an integer of value 0.
    - All the built-in types can be called with no arguments.
  - **2) a data type called with a single argument.**
    - If an argument of the same type is given, a new object which is a shallow copy of the original object is created.
    - If an argument of a different type is given, a conversion is attempted.
    - If the argument is of a type that supports conversions to the given type and the conversion fails, a ValueError exception is raised; otherwise, the resultant object of the given type is returned.
    - If the argument's data type does not support conversion to the given type, a TypeError exception is raised.
  - **3) a datatype called with two or more arguments**
    - not all types support this, and for those that do the argument types and their meanings vary.
    - For the int type two arguments are permitted where the first is a string that represents an integer and the second is the number base of the string representation.
      - For example, int("A4", 16) creates an integer of value 164.

Integer Conversion Functions

Syntax	Description
`bin(i)`	Returns the binary representation of `int i` as a string, e.g., `bin(1980) == '0b11110111100'`
`hex(i)`	Returns the hexadecimal representation of `i` as a string, e.g., `hex(1980) == '0x7bc'`
`int(x)`	Converts object `x` to an integer; raises `ValueError` on failure—or `TypeError` if `x`'s data type does not support integer conversion. If `x` is a floating-point number it is truncated.
`int(s, base)`	Converts `str s` to an integer; raises `ValueError` on failure. If the optional *`base`* argument is given it should be an integer between 2 and 36 inclusive.
`oct(i)`	Returns the octal representation of `i` as a string, e.g., `oct(1980) == '0o3674'`

Integer Bitwise Operators

Syntax	Description
`i | j`	Bitwise OR of `int i` and `int j`; negative numbers are assumed to be represented using 2's complement
`i ^ j`	Bitwise XOR (exclusive or) of `i` and `j`
`i & j`	Bitwise AND of `i` and `j`
`i << j`	Shifts `i` left by `j` bits; like `i * (2 ** j)` without overflow checking
`i >> j`	Shifts `i` right by `j` bits; like `i // (2 ** j)` without overflow checking
`~i`	Inverts `i`'s bits

Booleans

- There are two built-in Boolean objects:
  - True
  - False

```
>>> x = True
>>> x
True
>>> print(x)
True
```

```
>>> y = False
>>> y
False
>>> print(y)
False
```

- The bool data type can be called as a function
  - with no arguments it returns False
    - >>> bool()
      - False
  - with a bool argument it returns a copy of the argument
    - >>> bool(False)
      - False
  - with any other argument it attempts to convert the given object to a bool.
    - >>> bool(5)
      - True

Floating-Point Types

Floating-Point Types

- Python provides three kinds of floating-point values:
  - float
  - complex type
  - the decimal

- Numbers of type float are written with a decimal point, or using exponential notation.
  - Example: 0.0, 4., 5.7, -2.5, -2e9, 8.9e-4

Floating-Point Numbers

- The float type in Python assigns a floating-point number.
- float values are specified with a decimal point.
  - Optionally, the character e or E followed by a positive or negative integer can be added to specify scientific notation

- Example:

```
>>> 4.2
        4.2
>>> type(4.2)
        <class 'float'>
 >>> 4.
      4.0
>>> .2
        0.2
```

```
>>> import sys
>>> sys.float_info
```

Output:

```
sys.float_info(max=1.7976931348623157e+308,
max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-
1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2,
rounds=1)
```

DSCE
Dept. of Information Science & Engg

Floating-Point Numbers

- In addition to the built-in floating-point functionality, the math module provides many more functions that operate on floats.

Syntax	Description
`math.acos(x)`	Returns the arc cosine of `x` in radians
`math.acosh(x)`	Returns the arc hyperbolic cosine of `x` in radians
`math.asin(x)`	Returns the arc sine of `x` in radians
`math.asinh(x)`	Returns the arc hyperbolic sine of `x` in radians
`math.atan(x)`	Returns the arc tangent of `x` in radians
`math.atan2(y, x)`	Returns the arc tangent of `y / x` in radians
`math.atanh(x)`	Returns the arc hyperbolic tangent of `x` in radians
`math.ceil(x)`	Returns $\lceil x \rceil$, i.e., the smallest integer greater than or equal to `x` as an `int`; e.g., `math.ceil(5.4) == 6`
`math.copysign(x,y)`	Returns `x` with `y`'s sign
`math.cos(x)`	Returns the cosine of `x` in radians
`math.cosh(x)`	Returns the hyperbolic cosine of `x` in radians
`math.degrees(r)`	Converts `float r` from radians to degrees
`math.e`	The constant e; approximately 2.718 281 828 459 045 1
`math.exp(x)`	Returns e^x, i.e., `math.e ** x`
`math.fabs(x)`	Returns $\lvert x \rvert$, i.e., the absolute value of `x` as a `float`

The Math Module's Functions and Constants

DSCE
Dept. of Information Science & Engg

Floating-Point Numbers

`math.factorial(x)`	Returns $x!$
`math.floor(x)`	Returns $\lfloor x \rfloor$, i.e., the largest integer less than or equal to `x` as an `int`; e.g., `math.floor(5.4) == 5`
`math.fmod(x, y)`	Produces the modulus (remainder) of dividing `x` by `y`; this produces better results than `%` for `floats`
`math.frexp(x)`	Returns a 2-tuple with the mantissa (as a `float`) and the exponent (as an `int`) so, $x = m \times 2^e$; see `math.ldexp()`
`math.fsum(i)`	Returns the sum of the values in iterable `i` as a `float`
`math.hypot(x, y)`	Returns $\sqrt{x^2 + y^2}$
`math.isinf(x)`	Returns `True` if `float x` is $\pm$`inf` ($\pm\infty$)
`math.isnan(x)`	Returns `True` if `float x` is `nan` ("not a number")
`math.ldexp(m, e)`	Returns $m \times 2^e$; effectively the inverse of `math.frexp()`
`math.log(x, b)`	Returns $\log_b x$; b is optional and defaults to `math.e`
`math.log10(x)`	Returns $\log_{10} x$
`math.log1p(x)`	Returns $\log_e(1 + x)$; accurate even when x is close to 0
`math.modf(x)`	Returns `x`'s fractional and whole parts as two `floats`

The Math Module's Functions and Constants

Complex Numbers

- A complex number is of the form a + bi where a and b are real numbers, and i signifies the imaginary part, satisfying the equation $i^2 = -1$.

- Creating complex numbers in python

```
>>> c = 1 + 2j
>>> print(type(c))
>>> print(c)

>>> c1 = complex(2, 4)
>>> print(type(c1))
>>> print(c1)
```

Output:

```
<class 'complex'>
(1+2j)
<class 'complex'>
(2+4j)
```

Note: Every complex number contains one real part and one imaginary part.

Complex Numbers

- In general, complex() method takes two parameters:
  - **real** - real part. If real is omitted, it defaults to 0.
  - **imag** - imaginary part. If imag is omitted, it defaults to 0.

- If the first parameter passed to this method is a string, it will be interpreted as a complex number. In this case, the second parameter shouldn't be passed.

```
>>> z = complex(2, -3)
>>> print(z)
>>>z = complex(1)
>>> print(z)
>>> z = complex()
>>> print(z)
>>> z = complex('5-9j')
>>>print(z)
```

Output:

```
(2+3j)
(1+0j)
0j
(5-9j)
```

Python Complex Numbers Attributes and Functions

```
>>> c = 1 + 2j
>>> print('Real Part =', c.real)
>>> print('Imaginary Part =', c.imag)
>>> print('Complex conjugate =', c.conjugate())
```

Output:

Real Part = 1.0
Imaginary Part = 2.0
Complex conjugate = (1-2j)

Complex Numbers Mathematical Calculations

```
>>> c1= 1 + 2j
>>> c2 = 2 + 4j
>>> print('Addition =', c1 + c2)
>>> print('Subtraction =', c1 - c2)
>>> print('Multiplication =', c1* c2)
>>> print('Division =', c2 / c1)
```

Output:

Addition = (3+6j)
Subtraction = (-1-2j)
Multiplication = (-6+8j)
Division = (2+0j)

Example:

$(3 + 2\mathbf{i})(1 + 7\mathbf{i}) = 3\times1 + 3\times7\mathbf{i} + 2\mathbf{i}\times1 + 2\mathbf{i}\times7\mathbf{i}$

$= 3 + 21\mathbf{i} + 2\mathbf{i} + 14\mathbf{i}^2$

$= 3 + 21\mathbf{i} + 2\mathbf{i} - 14$ *(because* $\mathbf{i}^2 = -1$*)*

$= -11 + 23\mathbf{i}$

How do we divide complex numbers?

$$Ex: \frac{3+11i}{-1-2i} * \frac{-1+2i}{-1+2i}$$

$$= \frac{(3+11i)(-1+2i)}{(-1-2i)(-1+2i)}$$

$$= \frac{-3+6i-11i+22i^2}{1-2i+2i-4i^2}$$

$$= \frac{-3-5i+22(-1)}{1-4(-1)}$$

$$= \frac{-3-5i-22}{1+4}$$

$$= \frac{-25-5i}{5}$$

$$= \frac{-25}{5} - \frac{5i}{5}$$

$$= -5-i$$

Decimal Numbers

- Decimal numbers are just the floating-point numbers with fixed decimal points.
  - Python's decimal module helps us to be more precise with decimal numbers.

    Example:

```
>>> division = 72 / 7
>>> print(type(division))
 <class 'float'>
>>> print(division)
10.285714285714286
```

  - Calculations involving Decimals are slower than those involving floats
  - Use decimal type if the accuracy of rounding is preferred. With floats, accuracy related issues can be seen.

Using the decimal module

- Python built-in class float performs a few calculations that may surprise us.
  - Example:

```
>>> (1.1 + 2.2) == 3.3
False
>>> 1.1+2.2
3.3000000000000003
```

- Example:

```
>>> import decimal
>>># from decimal import Decimal (specific function from module)
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

Note: Decimal numbers are created using the decimal.Decimal() function. This function can taken an integer or a string argument but not float.

Using the decimal module

```
>>> import decimal
>>> division = decimal.Decimal(72) / decimal.Decimal(7)
>>> print(division)
10.285714285714285714285714
```

- Numbers of type decimal.Decimal work within the scope of a *context*
- *The* context is a collection of settings that affect how decimal.Decimals behave.

```
import decimal
with decimal.localcontext() as ctx:
   ctx.prec = 3
   division = decimal.Decimal(72) / decimal.Decimal(7)
   print(division)

again = decimal.Decimal(72) / decimal.Decimal(7)
print(again)
```

Output:

```
10.3
10.285714285714285714285714
```

The context specifies the precision that should be used

Getting Python Decimal Context

```
>>> from decimal import *
>>> print(getcontext())
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
    capitals=1, clamp=0, flags=[Inexact, FloatOperation, Rounded],
    traps=[InvalidOperation, DivisionByZero, Overflow])
```

Rounding the numbers

```
>>> rounded = round(13.485, 2)
>>> print(rounded)
13.48
```

Strings

Strings

- A string is a sequence of characters.
- Computers deal with characters in numbers (binary) form.
  - The translation of character to a number is called encoding, and the reverse process is called decoding.

    Ex: ASCII (American Standard Code for Information Interchange - defined unaccented characters with the numeric values running from 0 to 127) and Unicode (Describes how characters are represented by **code points.** A code point is an integer value, usually denoted in base 16 i.e 2^16 = 65,536 distinct values. Unicode includes every character in all languages.)
- In Python, a string is a sequence of Unicode characters.

Strings

- The str data type can be called as a function to create string objects:
  - with no arguments it returns an empty string
  - with a nonstring argument it returns the string form of the argument and
  - with a string argument it returns a copy of the string.
-
- The str() function can also be used as a conversion function
  - the first argument should be a string or something convertible to a string,
  - Next two - optional string arguments
    - one specifying the encoding to use and
    - the other specifying how to handle encoding errors.

Example:

```
>>> s1 = str()
>>> print(type(s1))
<class 'str'>
>>> print(s1)

>>> s2 = str(5)
>>> print(s2)
5
>>> s3 = str('my_str')
>>> print(s3)
my_str
```

How to create a string in Python?

- Strings can be created by enclosing characters inside a single quote or double-quotes.
- Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.
- Example:

```
>>> my_string = 'Hello'
>>> print(my_string)
Hello
>>> my_string = "Hello"
>>> print(my_string)
Hello
>>> my_string = '''Hello'''
>>> print(my_string)
Hello
>>> my_string = """Hello, welcome to
        Programming in Python """
>>> print(my_string)
Hello, welcome to
        Programming in Python
```

We can join two strings together by putting them side by side:

```
>>> my_string = "Python" " " Programming"
>>> print(my_string)
Python Programming
```

Python's String Escapes

- Quotes can be used inside a normal quoted string without regulation if they are different from the delimiting quotes; otherwise, quotes must be escaped.

- Example:
  - a = "Single 'quotes' are fine; \"doubles\" must be escaped."
  - b = 'Single \'quotes\' must be escaped; "doubles" are fine.'

- Newlines can be used without regulation in triple quoted strings

- Escape sequences are also used in writing regular expressions

DSCE
Dept. of Information Science & Engg

Python's String Escapes

Escape	Meaning
newline	Escape (i.e., ignore) the newline
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{*name*}	Unicode character with the given name
ooo	Character with the given octal value
\r	ASCII carriage return (CR)
\t	ASCII tab (TAB)
\u*hhhh*	Unicode character with the given 16-bit hexadecimal value
\U*hhhhhhhh*	Unicode character with the given 32-bit hexadecimal value
\v	ASCII vertical tab (VT)
\x*hh*	Character with the given 8-bit hexadecimal value

Python uses newline as its statement terminator, except inside parentheses (()), square brackets ([]), braces ({}), or triple quoted strings.

DSCE
Dept. of Information Science & Engg

Regular expression

- Regular Expression is a sequence of characters used to check whether a pattern exists in a given text (string) or not.

- Metacharacters : are characters with a special meaning:
  - \ Used to drop the special meaning of character following it
  - [] Represent a character class
  - ^ Matches the beginning
  - $ Matches the end . Matches any character except newline
  - ? Matches zero or one occurrence.
  - | OR - Matches with any of the characters separated by it.
  - * Any number of occurrences (including 0 occurrences)
  - + One or more occurrences
  - {} Indicates number of occurrences of a preceding RE to match.
  - () Enclose a group of Res

- In Python, regular expressions are supported by the re module.

Regular expression : Examples

```
import re
 # compile() creates regular expression character class [a-e]
# which is equivalent to [abcde].
# class [abcde] will match with string with 'a', 'b', 'c', 'd', 'e'.
p = re.compile('[a-e]')
 # findall() searches for the Regular Expression and return a list upon finding
print(p.findall("An apple a day is good for health"))
```

```
Output:
['a', 'e', 'a', 'd', 'a', 'd', 'e', 'a']
```

```
import re
# \d is equivalent to [0-9].
p = re.compile('\d')
print(p.findall("I went to collge at 11 A.M. on 15th Jan 2020"))
# \d+ will match a group on [0-9], group of one or greater size
p = re.compile('\d+')
print(p.findall("I went to collge at 11 A.M. on 15th Jan 2020"))
```

```
Output:

['1', '1', '1', '5', '2', '0', '2', '0']
['11', '15', '2020']
```

Unicode encoding

- .py files default to using the UTF-8 Unicode encoding.

- Unicode characters can be used inside strings using hexadecimal escape sequences or using Unicode names.
  - For example:

```
>>> euros = "€ \N{euro sign} \u20AC\U000020AC"
>>> print(euros)
€ € € €
```

- ord() function : to know the Character Unicode code point

```
>>> ord(euros[0])
8364
>>> hex(ord(euros[0]))
'0x20ac'
```

Unicode encoding

- any integer that represents a valid code point can be converted into the corresponding Unicode character using the built-in chr() function

- Example:

```
>>> s2 = "Euros symbol: " + chr(8364) + chr(0x20ac)
>>> s2
'Euros symbol: €€'

>>> s = "anarchists are " + chr(8734) + chr(0x23B7)
>>> s
'anarchists are ∞⎷'
```

Comparing Strings

- Strings support the usual comparison operators <, <=, ==, !=, >, and >=.
- Unfortunately, two problems arise when performing comparisons:

(1) Some Unicode characters can be represented by two or more different byte sequences.

– Example: the character *Å (Unicode code* point 0x00C5) UTF-8 encoded bytes : [0xE2, 0x84, 0xAB], [0xC3, 0x85], and [0x41, 0xCC, 0x8A].

» unicodedata.normalize() with "NFKC" as the first argument (this is a normalization method—three others are also available, "NFC", "NFD", and "NFKD"), and a string containing the *Å character using any of its valid byte sequences,* will return a string that when represented as UTF-8 encoded bytes will always be the byte sequence [0xC3, 0x85].

Comparing Strings contd…

(2) the sorting of some characters is language-specific.

- ø get sorted as if it were o in English, but in Danish and Norwegian it is sorted after z
- some characters (such as arrows, mathematical symbols etc..) don't really have meaningful sort positions.

Note: In python , string comparison is based on strings' in memory byte representation which produces English language ordering.

Slicing and Striding Strings

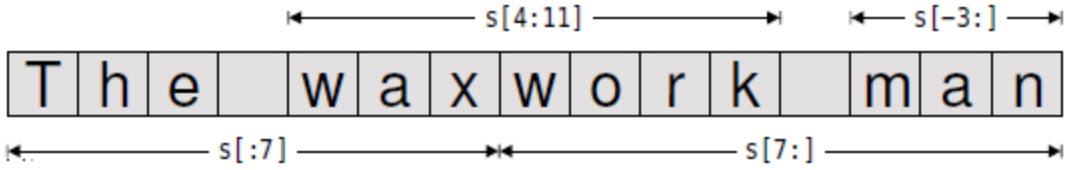
- Individual characters in a string can be extracted using the item access operator ([]). [] can be used to extract
  - a character or
  - an entire slice (subsequence) / characters, in which context it is referred to as the slice operator

- String indexing starts at **0 in the beginning of the string and work their way backward from -1 at the end.**

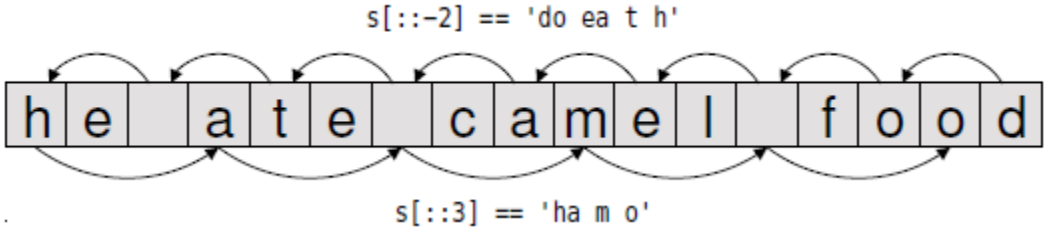
Slicing and Striding Strings contd…

- The slice operator has three syntaxes:
  - *seq[start]* - extracts the *start-th item from the sequence*
  - *seq[start:end] - extracts a slice from and including the start-th item, up to and excluding the end-th item*
    - If the start index is omitted, it will default to 0.
    - If the end index is omitted, it will default to len(*seq).*
  - *seq[start:end:step] - extracts* every *step-th character from and including the start-th item, up to and excluding the end-th item*
    - If the start index is omitted, it will default to 0. If a negative step is start index defaults to -1.
    - If the end index is omitted, it will default to len(seq). If a negative step is given, the end index defaults to before the beginning of the string.
- The seq can be any sequence, such as a list, string, or tuple.
- The start, end, and step values must all be integers (or variables holding integers).

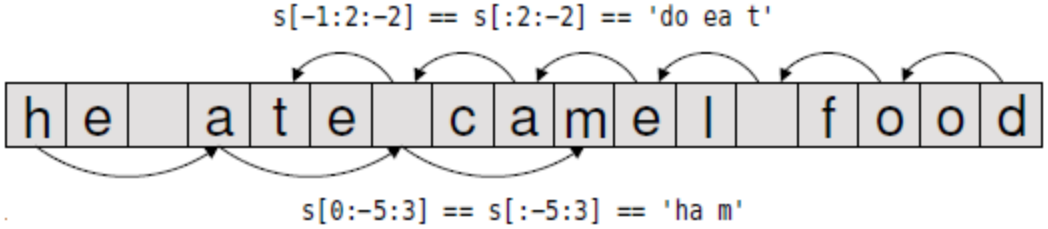
Slicing and Striding Strings : Examples contd...



Sequence slicing



Sequence striding



Sequence slicing and striding

Slicing and Striding Strings contd…

- One way of inserting a substring inside a string is to mix slicing with concatenation.
  - Ex:

```
>>> s = "The waxwork man"
>>> s = s[:12] + "wo" + s[12:]
>>> s
'The waxwork woman‘
```

- Since the text “wo” appears in the original string, the same effect can be achieved by assigning s[:12] + s[7:9] + s[12:]
  - Ex:

```
>>> s1 = "The waxwork man"
>>> s2 = s1[:12] + s1[7:9] + s1[12:]
>>> s2
'The waxwork woman‘
```

Using + to concatenate and += to append is not efficient when many strings are involved. For joining lots of strings it is usually best to use the str.join() method

Slicing and Striding Strings contd…

- Striding is most often used with sequence types other than strings.

- But Striding is preferred in one context for usage with strings:

```
>>> s2, s2[::-1]
('The waxwork woman', 'namow krowxaw ehT')
```

  - Stepping by -1 extracts every character from the end back to the beginning—and therefore produces the string in reverse.

String Operators and Methods

- All the functionality that can be used with immutable sequences can be used with strings

- Examples:

  - Membership testing with in
  - concatenation with +
  - appending with +=
  - replication with * and
  - augmented assignment replication with *=

String Operators

- **Concatenation (+) :** It combines two strings into one.
- Example :

```
var1 = 'Python'
var2 = 'String'
print (var1+var2)
```

PythonString

- **Repetition (*) :** This operator creates a new string by repeating it a given number of times.
- Example :

```
var1 = 'Python'
print (var1*3)
```

PythonPythonPython

```
s = "=" * 5
 print(s)
=====
s *= 10
print(s)
==================================================
```

String Operators contd..

- **Slicing [ ] :** The slice operator prints the character at a given index.
- Example :

```
var1 = 'Python'
print (var1[2])
T
```

- **Range Slicing [x:y] :** It prints the characters present in the given range.
- Example:

```
var1 = 'Python'
print (var1[2:5])
tho
```

String Operators contd..

- **Membership (in) :** This operator returns 'True' value if the character is present in the given String.
- Example:

```
var1 = 'Python'
print ('n' in var1)
True
```

- **Membership (not in) :** It returns 'True' value if the character is not present in the given String.
- Example :

```
var1 = 'Python'
print ('N' not in var1)
True
```

String Operators contd..

- **Iterating (for) :** With this operator, we can iterate through all the characters of a string.
- Example :

```
for var in var1:
    print (var, end ="")
Python
```

- **Raw String (r/R) :** it ignores the actual meaning of Escape characters inside a string. For this, add ‘r’ or ‘R’ in front of the String.
- Example:

```
print (r'\n')
\n
print (R'\n')
\n
```

String Functions

- **capitalize() –** Returns the string with the first character capitalized and rest of the characters in lower case.
- Example:

```
var = 'PYTHON'
print (var.capitalize())
Python
```

- **lower() –** Converts all the characters of the String to lowercase
- Example:

```
var = ‘DayanandaSagar'
print (var.lower())
dayanandasagar
```

String Functions contd…

- **upper() –** Converts all the characters of the String to uppercase
- Example:

```
var = ' DayanandaSagar'
print (var.upper())
DAYANANDASAGAR
```

- **swapcase() –** Swaps the case of every character in the String means that lowercase characters got converted to uppercase and vice-versa.
- Example:

```
var = ' DayanandaSagar '
print (var.swapcase())
dAYANANDAsAGAR
```

String Functions contd…

- **title() –** Returns the 'titlecased' version of String, which means that all words start with uppercase and the rest of the characters in words are in lowercase.

- Example:

```
var = 'welcome to Python programming'
print (var.title())
Welcome To Python Programming
```

String Functions contd…

- **count(str[, beg [, end]]) –** Returns the number of times substring 'str' occurs in the range [beg, end] if beg and end index are given else the search continues in full String Search is case-sensitive.

- Examples:

```
var='Dayanandasagar‘
str='a‘
print (var.count(str))
6

var1='Eagle Eyes'
print (var1.count('e'))
2

var2='Eagle Eyes'
print (var2.count('E',0,5))
 1
```

String Functions contd…

- To concatenate lots of strings use str.join(). The method takes a sequence as an argument (e.g., a list or tuple of strings), and joins them together into a single string.

- Example:

```
treatises = ["Arithmetica", "Conics", "Elements"]
" ".join(treatises)
'Arithmetica Conics Elements‘
"-<>-".join(treatises)
'Arithmetica-<>-Conics-<>-Elements‘
 "".join(treatises)
'ArithmeticaConicsElements'
```

String Functions contd…

METHOD	DESCRIPTION
str.capitalize()	Capitalizes first letter of string
len(str)	Returns the length of the string
str.lower()	Converts all uppercase letters in string to lowercase
str.upper()	Converts lowercase letters in string to uppercase
join(seq)	Merges(concatenates) the string representations of elements in sequence seq into a string, with separator string
str.count(str, beg= 0,end=len(string))	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
str.find(str, beg=0 end=len(string))	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
str.index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found.

String Functions contd…

METHOD	DESCRIPTION
str.isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
str.isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
str.isdigit()	Returns true if string contains only digits and false otherwise.
str.islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
str. isspace()	Returns true if string contains only whitespace characters and false otherwise.
str. isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.

String Functions contd…

METHOD	DESCRIPTION
str.lstrip([chars])	Removes all leading whitespace in string.
max(str)	Returns the max alphabetical character from the string str.
min(str)	Returns the min alphabetical character from the string str.
replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
str.rfind(str, beg=0,end=len(string))	Same as find(), but search backwards in string.
str.rindex(str, beg=0, end=len(string))	Same as index(), but search backwards in string.
str.rstrip()	Removes all trailing whitespace of string.
str.strip([chars])	Performs both lstrip() and rstrip() on string
str.swapcase()	Inverts case for all letters in string.

String Functions contd…

- str.index() method often produces cleaner code

```
def extract_from_tag(tag, line):
        opener = "<" + tag + ">"
        closer = "</" + tag + ">"
        try:
                i = line.index(opener)
                start = i + len(opener)
                 j = line.index(closer, start)
                return line[start:j]
        except ValueError:
                return None
```

```
def extract_from_tag(tag, line):
        opener = "<" + tag + ">“
        closer = "</" + tag + ">“
         i = line.find(opener)
         if i != -1:
                start = i + len(opener)
                 j = line.find(closer, start)
                 if j != -1:
                        return line[start:j]
         return None
```

Both versions of the extract_from_tag() function have exactly the same behavior.
For example, **extract_from_tag("red", "what a <red>rose</red> this is")** returns the string “**rose**”.

String Functions contd…

- **islower()** – Returns 'True' if all the characters in the String are in lowercase.
- Example:

```
>>> var='Python'
>>> print (var.islower())
False
```

```
>>> var='python'
>>> print (var.islower())
True
```

- **isupper()** – Returns 'True' if all the characters in the String are in uppercase.
- Example:

```
>>> var='Python'
>>> print (var.isupper())
 False
```

```
>>> var='PYTHON'
>>> print (var.isupper())
 True
```

String Functions contd…

- **isdecimal() –** Returns True if s is nonempty and every character in s is a Unicode base 10 digit
- Example:

```
>>> Num='2020'
>>> print (num.isdecimal())
True
```

- **isdigit() –** Returns True if s is nonempty and every character in s is an ASCII digit.
- Example:

```
>>> print ('2'.isdigit())
True
```

String Functions contd…

- **rjust(width[,fillchar]) –** Returns string filled with input char while pushing the original content on the right side.
  - By default, the padding uses a space. Otherwise, 'fillchar' specifies the filler character.
  - Example:

```
>>> var='Python'
>>> print (var.rjust(20))
            Python
```

- **ljust(width[,fillchar]) –** Returns a padded version of String with the original String left-justified to a total of width columns
  - By default, the padding uses a space. Otherwise, 'fillchar' specifies the filler character.
  - Example:

```
>>> var='Python'
>>> print (var.ljust(15,'-'))
Python---------
```

String Functions contd…

- **center(width[,fillchar]) –** Returns string filled with the input char while pushing the original content into the center.
  - By default, the padding uses a space. Otherwise, 'fillchar' specifies the filler character.
  - Example:

```
>>> var='Python'
>>> print (var.center(20))
      Python
>>> print (var.center(20,'*'))
*******Python*******
```

- **zfill(width) –** Returns string filled with the original content padded on the left with zeros so that the total length of String becomes equal to the input size.
  - If there is a leading sign (+/-) present in the String, then with this function, padding starts after the symbol, not before it.
  - Example:

```
>>> var='Python'
>>> print (var.zfill(10))
0000Python
>>> var1 = '+Python'
>>> print (var1.zfill(10))
+000Python
```

String Functions contd…

- s.find(t, *start, end) :* Returns the leftmost position of t in s (or in the *start:end* slice of s) or -1 if not found.

- Example:

```
>>> var="Dayananda sagar"
>>> str = "sagar"
>>> print (var.find(str))
10
>>> print (var.find(str,5))
10
>>> print (var.find(str,11))
-1
```

String Functions contd…

- s.index(t, *start, end) :* Returns the leftmost position of t in s (or in the *start:end slice of s) or raises ValueError if not found.*

  – *Example:*

```
>> var="Dayananda sagar"
>>> str = "sagar"
>>> print (var.index(str))
10
>>> print (var.index(str,5))
10
>>> print (var.index(str,11))
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    print (var.index(str,11))
ValueError: substring not found
```

String Functions contd…

- **rfind(str[,i [,j]]) –** This is same as find() just that this function returns the last index where ‘str’ is found. If ‘str’ is not found, it returns ‘-1’.

- Example:

```
>>> var='This is a good example'
>>> str='is'
>>> print (var.rfind(str,0,10))
5
>>> print (var.rfind(str1,5))
10
>>> print (var.rfind(str1,15))
-1
```

String Functions contd…

- **replace(old,new[,count]) –** Replaces all the occurrences of substring 'old' with 'new' in the String.
  - If the count is available, then only 'count' number of occurrences of 'old' will be replaced with the 'new' var.

- Example:

```
>>> var='This is a good example'
>>> str='was'
>>> print (var.replace('is',str))
Thwas was a good example
>>> print (var.replace('is',str,1))
Thwas is a good example
```

String Functions contd…

- **split([sep[,maxsplit]]) –** Returns a list of substring obtained after splitting the String with 'sep' as a delimiter.
  - Where, sep= delimiter, the default is space, maxsplit= number of splits to be done
- Example:

```
>>> var = "This is a good example"
>>> print (var.split())
['This', 'is', 'a', 'good', 'example']
>>> print (var.split(' ', 3))
['This', 'is', 'a', 'good example']
```

- **splitlines(num) –** Splits the String at line breaks and returns the list after removing the line breaks.
  - Where num = a positive value indicates that line breaks will appear in the returned list.
- Example:

```
>>> var='Print new line\nNextline\n\nMove again to new line'
>>> print (var.splitlines())
['Print new line', 'Nextline', '', 'Move again to new line']
>>> print (var.splitlines(1))
['Print new line\n', 'Nextline\n', '\n', 'Move again to new line']
```

String Functions contd…

- **s.strip(chars)** : Returns a copy of s with leading and trailing whitespace (or the characters in str chars) removed;str.lstrip() strips only at the start, and str.rstrip() strips only at the end

- Example:

```
>>> var='    This is a good example      '
>>> print (var.strip())
This is a good example
>>> var='*****This is a good example*****'
>>> print (var.strip('*'))
This is a good example
>>> print (var.lstrip('*'))
This is a good example*****
>>> print (var.rstrip('*'))
*****This is a good example
```

String Functions contd…

- **len(string) –** Returns the length of given String

- Example:

```
>>> var='This is a good example'
>>> print (len(var))
22
```

String Functions contd…

- **maketrans() :** returns a translation table that maps each character in the *inputstring* into the character at the same position in the *outputstring* . Then this table is passed to the translate() function.

- The **translate()** method returns a string where some specified characters are replaced with the character described in a dictionary/ mapping table.
  - str.maketrans(*inputstring*, *outputstring*) :
    - *Inputstring :* string having actual characters
    - *Outputstring :* string having corresponding mapping character.

```
Example:
>>> txt = "Hello Sam!“
>>> mytable = txt.maketrans("S", "R");
>>> print(txt.translate(mytable))
Hello Ram!

>>> inputstring = "aeiou"
>>> outputstring = "12345“
>>> str = "this is string example....wow!!!"
>>> mytable = str.maketrans(inputstring, outputstring)
>>> print (str.translate(mytable))
th3s 3s str3ng 2x1mpl2....w4w!!!
```

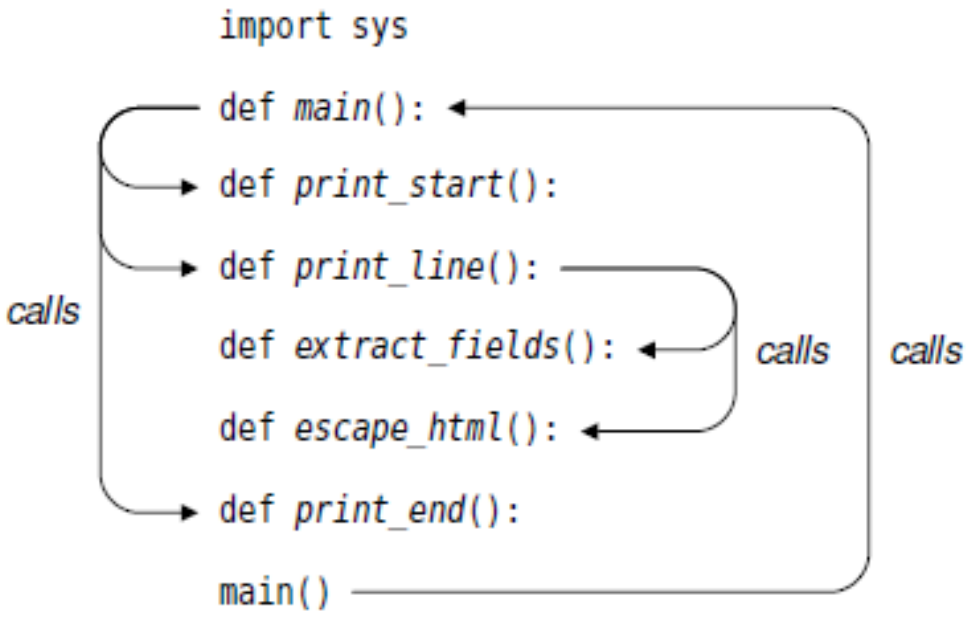
Example: quadratic.py

- The standard form of a quadratic equation is: $ax^2 + bx + c = 0$
  - where a, b and c are real numbers and
  - $a \neq 0$

```
import math

def equationroots( a, b, c):

    # calculating discriminant using formula
    dis = b * b - 4 * a * c
    sqrt_val = math.sqrt(abs(dis))

    # checking condition for discriminant
    if dis > 0:
        print(" real and different roots ")
        print((-b + sqrt_val)/(2 * a))
        print((-b - sqrt_val)/(2 * a))

    elif dis == 0:
        print(" real and same roots")
        print(-b / (2 * a))

    # when discriminant is less than 0
    else:
        print("Complex Roots")
        print(- b / (2 * a), " + i", sqrt_val)
        print(- b / (2 * a), " - i", sqrt_val)

# Driver Program
#a = 1
#b = 10
#c = -24
print("Quadratic function : (a * x^2) + b*x + c")
a = float(input("a: "))
b = float(input("b: "))
c = float(input("c: "))
# If a is 0, then incorrect equation
if a == 0:
        print("Input correct quadratic equation")

else:
    equationroots(a, b, c)
```

Example: csv2html.py

- Develop a program that reads a file that uses a simple CSV (Comma Separated Value) format and outputs an HTML table containing the file's data.



Program Structure

csv2html.py (using pandas)

```
import pandas as pd

# to read csv file named "samplee"
a = pd.read_csv("sample.csv")

# to save as html file
# named as "Table"
a.to_html("Table.htm")

# assign it to a
# variable (string)
html_file = a.to_html()
```

Output:

	COUNTRY	2000	2001	2002	2003	2004
0	ANTIGUA AND BARBUDA	0	0	0	0	0
1	ARGENTINA	37	35	33	36	39
2	BAHAMAS	1	1	1	1	1
3	BAHRAIN	5	6	6	6	6

csvtohtml.py

```
import sys

def main():
    maxwidth = 100
    print_start()
    count = 0
    while True:
        try:
            line = input()
            if count == 0:
                color = "lightgreen"
            elif count % 2:
                color = "white"
            else:
                color = "lightyellow"
            print_line(line, color, maxwidth)
            count += 1
        except EOFError:
            break
    print_end()
```

```
def print_start():
    print("<table border='1'>")
```

```
def print_line(line, color, maxwidth):
    print("<tr bgcolor='{0}'>".format(color))
    fields = extract_fields(line)
    for field in fields:
        if not field:
            print("<td></td>")
        else:
            number = field.replace(",", "")
            try:
                x = float(number)
                print("<td align='right'>{0:d}</td>".format(round(x)))
            except ValueError:
                field = field.title()
                field = field.replace(" And ", " and ")
                if len(field) <= maxwidth:
                    field = escape_html(field)
                else:
                    field = "{0} ...".format(
                            escape_html(field[:maxwidth]))
                print("<td>{0}</td>".format(field))
    print("</tr>")
```

```
def extract_fields(line):
    fields = []
    field = ""
    quote = None
    for c in line:
        if c in "\"'":
            if quote is None: # start of quoted string
                quote = c
            elif quote == c:  # end of quoted string
                quote = None
            else:
                field += c    # other quote inside quoted string
            continue
        if quote is None and c == ",": # end of a field
            fields.append(field)
            field = ""
        else:
            field += c       # accumulating a field
    if field:
        fields.append(field)  # adding the last field
    return fields
```

```
def escape_html(text):
    text = text.replace("&", "&amp;")
    text = text.replace("<", "&lt;")
    text = text.replace(">", "&gt;")
    return text


def print_end():
    print("</table>")


main()
```

Thank you