INST0004 Programming 2

Lecture 02: Programming with Variables, Numbers and Strings

Module Leader:
Dr Daniel Onah

2023-24

Welcome to INST0004 Programming 2

Copyright Note

Important information to adhere to

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce the concept of programming with Numbers and Strings in Python. At the end of the lecture, you should be able to:

- define and use variables and constants
- understand the properties and limitations of integers and floating-point numbers
- understand how to write arithmetic expressions and assignment statements
- appreciate the significance of in-built functions
- develop program that can read and process inputs and display the results
- learn how to use strings in Python
- appreciate the importance of good code layout and comments

Introduction to Variables

More on variables declaration

When we write a program, we will want to store values somewhere in memory so we could use them at some point within our program computations. In Python programming, we use **variables** to store values. Every variable that we create is connected to a value, which represents the information that is connected to the variable. In this section we will be covering variable declaration.

By definition, a **variable** is a named storage location in a computer program. Each variable we define or create has a name and holds a unique value. the Python interpreter associates a variable with the values it holds. For example, we can associate the context of a variable to that of a street with house numbers, each house in that street has it's own unique number and so on.

Introduction to Variables

More on variables declaration

Let's take a look at a simple example of variable declaration.

Syntax

```python
#!/usr/bin/env python
"""
A simple variable declaration
"""
# define a variable to hold a string value
message = "INST0004 Python Programming"
print(message)
```

Output

```
INST0004 Python Programming
```

Introduction to Variables

More on variables declaration

In creating a variable, it is perfectly legal to use the same name of a variable to form an assignment operation. The value of the variable would be computed and the new value will be stored in the variable. Let's look at an example.

Syntax

```python
#!/usr/bin/env python
"""
A simple variable declaration
"""
# define a variable to hold an integer value and assign the variable to
    another variable with the same name
amount = 12
print("Initial amount: ", amount)
amount = amount + 15
print("New amount: ", amount)
```

Introduction to Variables
More on variables declaration

Here you notice that the value of the first variable **amount** is displayed as 12. In the next construct, we then added 15 to the variable **amount** which then displays 27. What this means is that the value of the variable amount has now been updated or overridden by the new value. The memory location of the variable amount will be updated with the new value.

Output

```
Initial amount: 12
New amount: 27
```

Number Types

Manipulating data type

We use computer to manipulate data values that represent information and these values can be of different data types. In Python programming, each value is of a specific data type. The **data type** of a value expresses how the data is being manipulated in the computer and what kind of operations can be performed on the data. The most commonly used data type in programming is known as the primitive data type. Python uses quite a few data types which could be applied for storing **numbers**, **strings**, **files**, **containers** and so on. In Python, numbers can be represented in different forms or types, **integer**, **floating-point** etc. An **integer** value is a whole number without a fractional part. We usually flag this with a built-in ***int*** method. The *int* method is used to convert numbers to integer. When we used a fractional part such as 12.34, then this type is known as **floating-point** numbers and we used a built-in ***float*** method.

Number Types
Manipulating data type

When a number type such as integer or float-point numbers are used in Python programming, this is called a **number literal**. In Python, **when a number literal has a decimal point, it is a floating-point number literal, otherwise, it is an integer number literal**. A variable in Python can store a value of any type. Ideally, the data type is associated with the value, and not the variable. Therefore, it is very important to know the correct data type (e.g., ***int***, ***float*** etc) for each value that would be stored in the variable.

Number Types

Manipulating data type

For example, let's consider a variable that is initialized with a value of type *int*. The same variable can later hold a value of type *float*.

Syntax

```python
#!/usr/bin/env python
"""
A simple variable declaration
"""
# define a variable to hold an integer value and assign a floating-point
    value to another variable with the same name
interestRate = 2 # integer literal
interestRate = 2.5 # floating-point literal
```

Number Types

Manipulating data type

Number Literals in Python

- **3** *int* - An integer has no fractional part.
- **-3** *int* - Integers can be negative.
- **0** *int* - Zero is an integer.
- **0.3** *float* - A number with a fractional part has type float.
- **3.0** *float* - An integer with a fractional part. 0 has type float.
- **1E6** *float* - A number in exponential notation: **1 x 10^{6}or** ***1000000***.
  Numbers in exponential notation always have type **float**.

Number Types
Manipulating data type

Number Literals in Python

- **2.96E-2** *float* - Negative exponent: $\mathbf{2.96 \times 10^{-2}} = 2.96/100 = 0.0296$.
- **100,000** *Error* - Do not use a comma as a decimal separator.
- **3 1/2** *Error* - Do not use fractions, used decimal notation: **3.5** instead.

However, we should be careful on how we use a variable attributed to another data type. For instance, if you use variable containing a value of an unexpected type, then an error will occur in your program. Instead, once you have initialised a variable with a value of a particular type, ensure you take care that you keep storing values of the same type into that variable. A variable declared as integer primitive type can not store a **string literal**.

Variable Naming Convention

Declaring variable names

When we define a variable, we give it a name that explains its purpose. Whenever you define any variable name in Python, should follow these simple rules:

- Variable names should start with letter or underscore (_) character, and the remaining characters should be **letters**, **numbers**, or **underscore**.
- You should not use symbols such as **?** or **%**.
- You should not use **spaces** inside the names either.

Variable Naming Convention

Declaring variable names

- You can use uppercase letters to denote word boundaries, for example ***bottlePack***

Constant Variable

What is a constant variable?

A constant variable or simply a **constant** is a variable whose value should not be changed after it has been assigned an initial value. Some programming languages adhered to the rule of a constant variable and will generate a syntax error if you attempt to assign a new value to the variable. Constants variable are declared using **ALL** caps. When we define a constant variable using all capital letters, the value assigned to that variable cannot be changed. Let's look at an example of a constant variable:

```python
#!/usr/bin/env python
"""
Declaring a constant variable
"""
# define a constant variable
VOLUME = 3.0
SIZE = 50
```

Constant Variable

What is a constant variable?

Using this naming convention for a constant variable, you declare that you wish to use the variable in all capital letters to be a constant throughout the program. It is a best practice in programming to name constants in your program to illustrate that the values for example numeric that is stored in the variable. Let's look at an example to help us understand the significance of the constant variable.

```
1 #!/usr/bin/env python
2 """
3 A simple arithmetic operation
4 """
5 # using a constant variable
6 total_Volume = bottles * 3.0
7 total_Volume = bottles * VOLUME
```

In the program above, the significance of number **3.0** might shown clearly (in line 6). However, the second construct (in line 7) with the name constant VOLUME, makes the computation much clear.

Magic Number

What is a magic number

A magic number is a numeric constant that appears in your code without explanation. For example,

```python
#!/usr/bin/env python
"""
Using magic number
"""
totalVolume = bottles * 2
```

Why 2? Are bottles twice as voluminous as cans? No, the reason is that every bottle contains 2 liters. Use a named constant to make the code self-documenting:

```python
#!/usr/bin/env python
"""
Assign the magic number to a constant
"""
BOTTLE_VOLUME = 2.0
totalVolume = bottles * BOTTLE_VOLUME
```

Magic Number

What is a magic number

There is another reason for using named constants. Suppose We prefer programs that circumstances change, and the bottle volume is now 1.5 liters. If you used a named constant, you make a single change, and over those that appear you are done. Otherwise, you have to look at every value of 2 in your program and ponder whether it meant a bottle volume or something else. In a program that is more than a few pages long, that is incredibly tedious and error-prone. We prefer programs that are easy to understand, rather than those that work by magic.

Common Errors

Come errors with variable declaration

A variable must be created and initialized before it can be used for the first time within a program. For example, the following sequence of statements would not be legal in a Python program.

```
#!/usr/bin/env python
"""
Common errors in variable declaration
"""
VOLUME = 3.0 * Size # Error: Size has not yet been created.
Size = 5.0
```

In Python program, the statements are executed line-by-line in order. When the first statement is executed by the interpreter, it does not know that **Size** would be created in the next line, and it reports an **"undefined name"** error. The solution is to reorder the statements so that each variable used in the assignment expression is created and initialized before it is used.

Program Comments

Why is comment important

As you write your programs and this becomes more complex, you should add comments, explanations for human readers of your code. For example, here is a comment that explains the value used in a constant. This help us understand the significance of the constant variable.

```
#!/usr/bin/env python
"""
Add comment
"""
CAN_VOLUME = 0.355 # Liters in a 12-ounce can
```

Program Comments
Why is comment important

This comment explains the significance of the value 0.355 to a human reader. The interpreter does not execute comments at all within our code. Comment will ignore everything from after hashtag symbol (#) or delimiter to the end of the line. We can also use triple quotes (""" """) for **multi-line** comments as well.

It is a good practice to provide comments in your code. This helps other people or programmers who read your code understand your intent and the various constructs. In addition, you will find comments helpful when you review your own programs. Provide a comment at the top of your source file that explains the purpose of the program.

```python
#!/usr/bin/env python
"""
This is a triple quote for multi-line comments
"""
# This program computes the volume of a can
# and the total volume of six-pack of can
```

Arithmetic Operations
Basic arithmetic operations

Just as with other programming languages, Python also supports four basic arithmetic operations: **addition**, **subtraction**, **Multiplication** and **Division**. But Python uses different symbols for Multiplication (*) and Division (/).
In Python, you write n * m to denote multiplication. Similarly, division is also indicated as n $/m$.
For example,

$$x = \frac{n+m}{2}$$

In **Python programming** this will be expressed as:

$$x = (n+m)/2$$

Arithmetic Operations
Basic arithmetic operations

The symbols + - * / in the arithmetic operations are called **operators**. The combination of variables, literals, operators, and parentheses is known as **expression**. For example, an expression on the right-hand side of the equality sign is assigned to a variable in **Python** as illustrated below:

$$x = (n + m)/2$$

In Python, we use **parentheses** to indicate in which order the parts of the expression should be computed. For example, in the expression (n + m) / 2, the sum n + m is computed first, and then the sum is divided by 2. In contrast to the expression; n + m / 2, here it is only **m** that is divided by 2 and then the result sum by adding **n**.

Arithmetic Operations

Basic arithmetic operations

As in regular algebraic notation, multiplication and **division** have a **higher precedence** than **addition** and **subtraction**.

For example, in the expression **n + m / 2**, the division operation / is carried out first, even though the + operation occurs further to the **left**. In Python, similar to algebra, operators with the same precedence are executed from left-to-right.

For example,

15 - 6 - 3 would be **9 − 3** or the computed result as 6.

- If you mix integer and floating-point values in an arithmetic expression, the result would be a floating-point value.

For example, **5 + 2.0** would result to be a floating-point value of **7.0**

Python Powers Operator & Expression

Using raise to power in Python

In Python, the **power expression** is indicated by using the exponential operator ****** to denote the **power operation**. The ****** operator is used in Python to raise the number on the **left** to the power of the exponent of the **right**.
For example, in the expression **12 ** 2** , the number **12** is being raised to the the power of **2** (raised to the 2nd power. In **mathematics**, this *expression is often rendered* as 12^2, and what this simply means is that the number 12 is being multiplied by itself **twice** (or multiplied 2 times).

- Note that there can be no space between the two asterisks **(**)**
- In Python, similar to mathematics, the exponential operator has a **higher order of precedence** *than the other arithmetic operators*

Python Powers Operator: More & Expression

Using raise to power in Python

Let's look at some examples,

15 * 5 ** 3 is $15 * 5^3 = 1876$

Unlike the other arithmetic operators, power operators are evaluated from **right to left**. Thus, the **Python expression** 10 ** 2 * 3 is equivalent to $10^2 * 3 = 300$. In algebra, we use fractions and exponents to arrange expressions in a compact two-dimensional form. *But in Python, you have to write all expressions in a linear arrangement*.

For **example**, the mathematical expression here:

$$b \times \left(1 + \frac{r}{100}\right)^n$$

In **Python**, this expression becomes:

b * (1 + r / 100) ** n

Floor Division

Floor division in Python

When you divide two integers with the **division** (/ operator, you get a floating-point value). For example, **9/5** would yield 1.8.
In **floor division**, we discard the fractional part and compute the quotient of two integer numbers. **Floor division** is computed using the symbol **(//)**.
For example, the computing the floor division of **9//5**, will be evaluated to **1** as the result. In the early calculation we had **1.8** using the division operator. But in the case of the floor division, the fractional part **.8** is discarded. Let's see a floor division example in **mathematics**:

$$x = \left\lceil \frac{7}{2} \right\rceil * \frac{5}{3} \equiv 5$$

Translating the above equation to **Python** floor division:

```python
"""Floor division and multiplication"""
floorDivisionValue = 7//2 * 5/3
```

Floor Division
Exercise

Problem Statement

Simulate a postage stamp vending machine. A customer inserts pounds bills into the vending machine and then pushes a "purchase" button. The vending machine gives out as many **first-class stamps** as the customer paid for, and returns the **change in pence** stamps. Suppose a first-class stamp cost **31** pence.

Floor Division

Exercise

Write a program to simulates a stamp machine that receives bills in pounds and dispenses how many first class and pence stamps to the customer.

```python
"""A program to simulate a stamp machine operation"""
# define the price of a stamp in pence.
FIRST_CLASS_STAMP_PRICE = 31
# obtain the number of pounds.
pounds = input("Enter number of pounds: ") # input as string
pounds = int(pounds) # convert input to int, by type casting
# compute and print the number of stamps to dispense.
firstClassStamps = 100 * pounds // FIRST_CLASS_STAMP_PRICE #
    firstclassstamps X 0.44
change = 100 * pounds - firstClassStamps * FIRST_CLASS_STAMP_PRICE
# to check for the total pounds entered = (firstClassStamps x 0.31) +
    change
print("First class stamps: %6d" % firstClassStamps)
print("Pence stamps: %6d" % change)
```

Floor Division

Solution

Output

```
Enter number of pounds: 5
First class stamps:     16
Pence stamps:      4
```

Remainder Operation
Remainder in Python

The **remainder** operator is the opposite of the **floor division**. In the case of the remainder operation we *consider the fractional part and discard the integer values*. If we are interested in the remainder of a floor division, we have to use the symbol **%** operator in Python. Let's look at an example of how remainder operator is presented mathematically:

$$x = \frac{27}{5} \equiv 5R2$$

This can also be represented mathematically as:

$$x = 27 mod 5 \equiv 2$$

Translating the above equation to **Python** remainder expression:

```python
"""Remainder operation"""
remainderValue = 27 % 5
```

Floor Division & Remainder

Combining floor division and remainder in Python

We can combine the floor division and the remainder operation to compute a given number. **Note that the remainder operation is related to division**. The operator % is called **modulus**. It can also be called *modulo* or *mod.*, this has no relationship with the percentage operation. Let's take a look at a typical example of how to use // and % operations in Python. Suppose we have an account balance of **4589 *pence*** in a bank and we want to calculate the amount in pounds and pence.

Floor Division & Remainder

Combining floor division and remainder in Python

```python
#!/usr/nin/env python
"""
A program combining floor division and remainder operation
"""
pence = 4589

# compute the floor division for pounds
pounds = pence // 100 # set pounds to 45 and ignore remainder

# compute the remainder for pence
pence = pence % 100 # set pence to 89 and ignore floor division

# display the amount in pounds and pence
print("The account balance is:   {} and {} pence".format(pounds, pence))
print("The account balance is:   {}.{}".format(pounds, pence))
```

Combination of Assignment and Arithmetic

Combining assignment and arithmetic operations

In Python, you can **combine arithmetic and assignment operations**. For example, the instructions below illustrates this:

total += cans

This can be represented in another was as:

total = total + cans

Another similar example:

total *= 2

This can be written in another way as:

total = total * 2

When incrementing or decrementing by 1:

count += 1 or **count -= 1**

Joining Program Line

Joining and breaking long expression

If you have an expression that is too long to fit into a single line, you can continue it on another line provided the line break occurs inside parentheses or after a comma etc. For example,

```
x1 = ((-b + sqrt(b ** 2 - 4 * a * c))
/ (2 * a)) # This would be fine
```

However, if you omit the outermost parentheses, this would result to an error:

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c))
/ (2 * a) # This would produce an Error
```

The first line is a complete statement, which the Python interpreter processes. But, the next line, **/ (2 * a)**, **makes no sense by itself** as this is not part of the full expression.

Joining Program Line

Joining long expression

There is another form of joining long lines. If the last character of a line is a backslash, the line is joined with the one following it:

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c)) \
/ (2 * a) # This would be fine
```

However, you should be very careful not to put any spaces or tabs after the **backslash**.

Breaking Program Line

Breaking long expression

You can also break a long expression or code in your program after a **comma**. This means you program code continues to the next line without any issue. Let's look at an example:

```python
#!/usr/bin/env python
"""
A program to demonstrate breaking a long expression
"""
number1 = 23
number2 = 25
product = number1 * number2
print("The product of {} and {} is: {}".format(number1, number2,
        product)) # This would be fine
```

Function Calling
Calling a function in Python

By definition, a ***function is collection of programming instructions or constructs that perform a particular task***. We have been utilising the print() function to display information about our program, but there are many functions in **Python** which we would look at in this module. In this section, we will be looking at functions that operate with numbers.

Ideally, most functions in Python returns a value. When a function completes a given task, it passes or returns a value to the point where the function was called or invoked. For example the **abs()** function *returns the absolute value of a given number by returning the inverse of the sign of the number*. If a number has a negative sign — before the value, this function would return a positive + value of the numerical argument.

Function Calling : abs()

Calling a function in Python

When a function is called, the code inside the function is executed when the function is invoked. Let's look at an example of a simple **built-in** function call. For example, if we call an ***abs()*** function with a negative value **-105**, this would return the value **105**.

```python
#!/usr/nin/env python
"""
A program using a built-in abs() function
"""
number = -105
absoluteValue = abs(number)
print(absoluteValue)
```

Function Calling : abs()

Calling a function in Python

Ideally, the returned value from within a function can be used anywhere that a value of the same type can be used:

```python
#!/usr/nin/env python
"""
A program using a built-in abs() function
"""
number = -105
print(abs(number))
```

The **abs()** function requires **data input** to perform its task, in this case the **number (-105)** from which we compute the absolute value. As seen above, *data input that you passed or provide to a function are the arguments of the call*.

Function Calling : abs()

Calling a function in Python

When calling a function, *you must provide the correct number of arguments*. For example, note as the **abs()** function in the previous program takes exactly one argument. If you present a function call to the **abs()** function with arguments such as the example below:

```python
#!/usr/nin/env python
"""
A program using a built-in abs() function error output
"""
number = -105
print(abs(number, 2)) # TypeError
# OR
print(abs())# TypeError
```

These would produce as *TypeError: abs() takes exactly one argument (0 given)* and *TypeError: abs() takes exactly one argument (2 given)* respectively.

Function Calling : round()

Calling a function in Python

There are some built-in functions that could accept optional arguments that you could only provide in certain situations. An example is the round() function. When called with one argument, the function will return the nearest integer. Let's look at an example of calling a **round()** on a floating-point number.

```python
#!/usr/bin/env python
"""
A program to demonstarte callling a round() function
"""
number = 0.5678
print(round(number))
```

The program above will return the value ***1*** as the nearest integer.

Function Calling : round()

Calling a function in Python

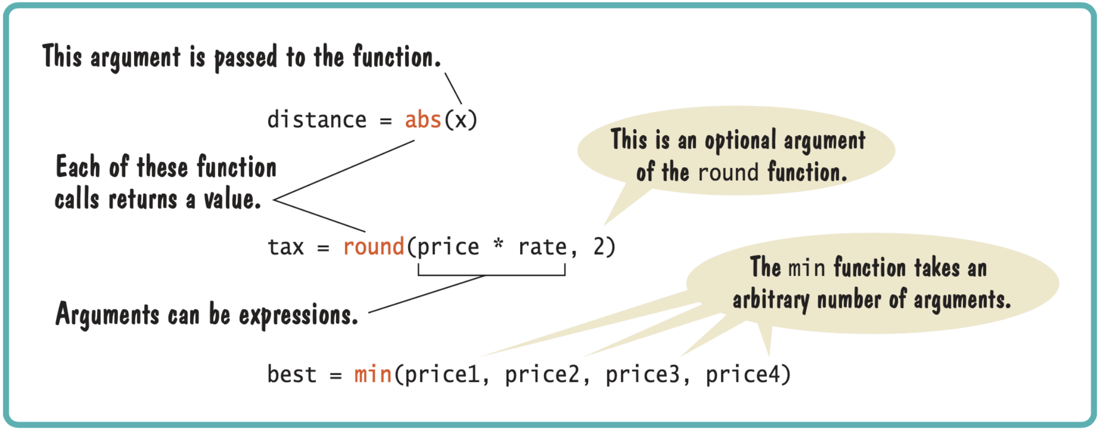
We can also call a **round()** function with two arguments, in this case, the second argument specifies the desired number of fractional digits. Let's say we are calling floating-point number, and we want to display the result in specific decimal places.

```python
#!/usr/bin/env python
"""
A program to demonstarte callling a round() function using two arguments
"""
number = 19.9345
print(round(number, 2))
```

The program above will return the value ***19.93*** in two decimal places. If we have value such as **19.9355** and we call a **round()** function with the second argument as 2, the result would be **19.94**.

Function Calling : abs() & round()

Calling a function in Python

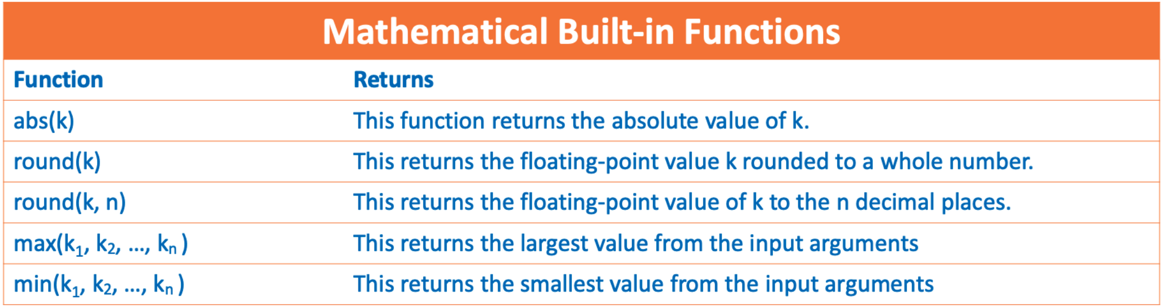

This argument is passed to the function.
Each of these function calls returns a value.
This is an optional argument of the round function.
Arguments can be expressions.
The min function takes an arbitrary number of arguments.

```
distance = abs(x)

tax = round(price * rate, 2)

best = min(price1, price2, price3, price4)
```

Image:Python for Everyone by ***Cay Horstman & Rance Necaise***

Built-in Functions

Mathematical built-in functions



Mathematical Built-in Functions	
Function	**Returns**
abs(k)	This function returns the absolute value of k.
round(k)	This returns the floating-point value k rounded to a whole number.
round(k, n)	This returns the floating-point value of k to the n decimal places.
max(k_1, k_2, ..., k_n)	This returns the largest value from the input arguments
min(k_1, k_2, ..., k_n)	This returns the smallest value from the input arguments

Built-in Functions

Mathematical built-in functions

Some built-in function such as **min** and **max** can take arbitrary number of arguments. For example:

```python
#!/usr/bin/env python
"""
A program to computer min() using arbitrary number of arguments
"""
smallest = min(78, 34, 8, 29, 90, 45, 12)
print(smallest)
```

This program will display the smallest number as **8**.

Python Libraries & Modules

Python programming contains standard library that can be used to create useful support to your programs. A **library** by *definition is a collection of code written and translated to use in a program*. A **standard library** are those that are considered as part of the language and must always be included within the Python system. Python Standard Library contains the exact syntax, semantics, and tokens of used in Python. This contains **built-in modules** that provide us with access to basic system functionality such as Input/Output functionalities and some other core modules. The current Python Libraries are written in the **C** programming language.

Python Libraries & Modules

Importing Libraries

You can import multiple functions from the same module as follows:

from math import sqrt, sin, cos

You can also import the entire contents of a module into your program:

from math import *

Let's look at an example of importing a **math** library:

```python
 #!/usr/bin/env python
"""
A program to demonstrate importing a math library
"""
import math
number = input("Enter number: ")
print(math.sqrt(number))
```

Python Libraries & Modules

Importing Libraries

We can import a module statement from a library by invoking or calling the function using the module name and a period **(.)** before each function call, like this:
number = math.sqrt(x) or **name = random.choice(x)**
Let's look at an example of importing a **random** library:

```python
#!/usr/bin/env python
"""
A program to demonstrate random library
"""
import random
name = ["Danny", "Onah"]
randName = random.choice(name) # choice:- for random string generation
print(randName)
```

Python Libraries & Modules

As we begin to write large-size programs with lots of algorithms in Python, we will want to also maintain the code's modularity. Program modularity allows us to split out codes into smaller chunks for easy maintenance of the code structure. In this case we **split our program code into different parts or smaller units** so we could use the code later whenever we need to. In Python programming, **modules** is used or allow us to play that vital role of modularity of our program code. Instead of using the **same code in different programs and making the code complex**, we define mostly used functions in modules and we can just simply import them into our program wherever there is a requirement for it. We don't need to write that code but still, we can use its functionality by **importing its module** into our programming listing or code.

Python Libraries & Modules

Libraries & Modules

Multiple interrelated modules are stored in the same library. Whenever we need to use a module, we just have to import the module **from** its library. In Python, this becomes very easy to execute because of less-complex syntax. All we need to do here is to use the **import** keyword.

Let's have a look at an example:

```python
 #!/usr/bin/evn python
"""
A program to demonstrate importing module from its library
"""
from math import sqrt
number = int(input("Enter number: "))
numb = sqrt(number)
print(numb)
```

Python Libraries & Modules

Revisiting main points

- Python's standard libraries are organized into modules.
- Related functions and data types are grouped into the same module.
- **Functions defined in a module must be explicitly loaded (or imported) into your program before they can be used.**
- In Python, the **math** module contains a number of mathematical functions.
- ***In order to use any function from Python library or module, you must first import the function.***

For example, to use the **sqrt** function, which computes the square root of its argument, first include the statement at the top of your program file (**import math** or **from math import sqrt**). Before you can call the function to perform the given task.

Re: Cap☺

Strings in Python

Python Strings

String library provides a set of methods and constants for working with string data. Strings are character sequences that are a fundamental **data type** in most programming languages. Several programs that we write process **text** and others compute **numbers**. Text consists of **characters** which could be in form of letters, numbers, punctuation, spaces, and so on. By definition, a String is a sequence of characters in Python programming. For example, the word or string *"HelloWorld"* is a sequence of **ten** characters. **Strings** are reference types and they can be stored in a **variable** and displayed using a print() function.
A ***string literal*** denotes a particular string, for example *"HelloWorld"* and ***number literal*** such as *2* denotes a certain integer number. In Python, we denote string literals by by enclosing a sequence of characters within a matching pair of either in a **single** **('')** or **double** **("")** quotes. For example:
print("This is a double quote string.", 'This is a single quote string.')

Strings in Python

Python Strings

We will be using double quotation marks around strings mostly in this lecture because this is a common convention in many other similar programming languages. Some languages uses single quote to denote characters mostly. However, for interactive Python interpreter this always displays strings with single quotation. We might get a SyntaxError if using a **double quotation** marks in an interactive Python interpreter.
Let's have a look at an example:

```
 #!/usr/bin/evn python
"""
A program error using double quote in Python interactive interpreter
"""
>>> greeting = "He said "Hello""
  File "<stdin>", line 1
    mess = "He said "Hello""
                     ^
SyntaxError: invalid syntax
```

Strings in Python
Python Strings

We can re-write the previous program using a single backslash (\). Let's have a look at an example:

```
 #!/usr/bin/evn python
"""
A program using single quote in Python interactive interpreter
"""
>>> greeting = "He said \"Hello\""
>>> print(greeting)
He said "Hello"
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence \" is called an **escape sequence**.

Strings in Python

Python Strings

However, the below **single quote** when use with Python interactive mode will work perfectly. Let's have a look at an example:

```
 #!/usr/bin/evn python
"""
A program using single quote in Python interactive interpreter
"""
>>> greeting = 'He said "Hello"'
>>> print(mess)
He said "Hello"
```

Length of a String: len()

String length

The number of characters in a string is called the length of the string. For example, the **length** of the name *"Danny"* is **5**. You can compute the length of a string using Python's built-in ***len()*** function. Let's have a look at an example:

```
 #!/usr/bin/evn python
"""
A program demonstrating the len() of a string
"""
length = len("Danny!") # length is 6
```

A string of length **0** is called an empty string. Such a string contains no characters and this is written and declare as (“”) or (‘’).

Concatenation & Repetition

String length

Given two names **“Danny”** and **“Onah”**, you can concatenate the names to one long single string. The output would consists of **all characters in the first name string**, followed by **all characters in the second name string**. In Python, we use the + operator to concatenate the two name strings. Let’s have a look at an example:

```python
 #!/usr/bin/evn python
"""
A program demonstrating string concatenation
"""
firstName = "Danny"
lastName = "Onah"
name = firstName + lastName
```

Output

```
DannyOnah
```

Concatenation & Repetition

String length

We can separate the first and last name with a space. Let's look at an example.

```
 #!/usr/bin/evn python
"""
A program demonstrating string concatenation
"""
firstName = "Danny"
lastName = "Onah"
name = firstName + " " + lastName
```

Output

```
Danny Onah
```

This statement concatenates three strings: **firstName**, the **string literal** " ", and the **lastName**.

Concatenation & Repetition

String length

When the expression to the left or the right of a + operator is a string, the other one must also be a string or a syntax error will occur. You **cannot concatenate a string with a numerical value**.

We can also produce a string as a result of repeating the string multiple times. For example, suppose you want display a dashed (-) line multiple times. Instead of specifying a literal string with **5** dashes, you can use the multiplication **(*)** operator to create a long string that would be comprised of the string "-" repeated 5 times. Let's look at an example.

```
 #!/usr/bin/evn python
"""
A program demonstrating string repetition
"""
dashes = "-" * 5
```

Concatenation & Repetition

String length

A string of any length can also be repeated using the * operator. Let's look at an example.

```
#!/usr/bin/evn python
"""
A program demonstrating string repetition
"""
greeting = "Hello..."
print(greeting * 5)
```

Output

```
Hello...Hello...Hello...Hello...Hello...
```

The factor by which the string is replicated or multiplied with must be an **integer value**. The **factor** can appear on either side of the * operator, but it is a common practice to place the string on the **left side** and the integer factor on the **right**.

Converting Numbers to String

Number to String

At some point, we might want to convert a numerical value to a string if this becomes necessary. Let say we want to append a number to the end of a string. We cannot concatenate a string literal and a number literal together. If we do this, it would lead to error Let's look at an example.

```
 #!/usr/bin/evn python
"""
A program demonstrating concatenating string and number
"""
name = "Danny" + 1729 # Error: Can only concatenate strings
print(name)
```

Converting Numbers to String

Number to String

String concatenation can only be performed between two strings. In order to concatenate a string and a number, we should first convert the number to a string. To produce the string representation of a numerical value, we use the ***str()*** function. Here we will need to convert the integer value 1729 to string "1729". The **str()** function help us to solve this problem. Let's look at an example.

```python
 #!/usr/bin/evn python
"""
A program demonstrating concatenating string and number
"""
number = 1729
name = "Danny " + str(number) # converting number to string using str()
    built-in function
```

Converting Strings to Numbers

String to Numbers

The **str()** function can also be used to convert a floating-point value to a string. Conversely, to convert a string containing a number into a numerical value, we have to use the int and float functions.

Let's look at an example.

```
 #!/usr/bin/evn python
"""
A program demonstrating converting string to numbers
"""
number = int("1729") # converting string to integer
amount = float("17.29") # converting string to floating-point
```

Converting Strings to Numbers

String to Numbers

String conversion is very important especially when the strings come from user input. The string passed to the **int** or **float** functions can only consist of those characters that comprise a literal value of the indicated data type. For example, let's look at an example statement:

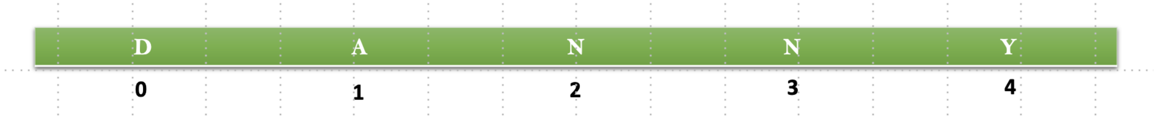
```
 #!/usr/bin/evn python
"""
A program demonstrating converting string to numbers
"""
value = float("17x29")  # run-time error
number = int(" 1729 ")  # no error, display integer value
```

This program construct will generate a **run-time error** because the letter **"x"** cannot be part of a floating-point literal. Blank spaces at the front or back will be ignored: int(" 1729 ") this will still display the output as **1729**.

Strings and Characters

Strings & characters

Strings are sequence of Unicode characters. **Individual characters** of a string can be accessed based on their position within the string. This **position** is known as the ***index*** of the character. The first character of every string has **index 0**, the second has **index 1**, and so on.



Strings and Characters

Strings & characters subscript

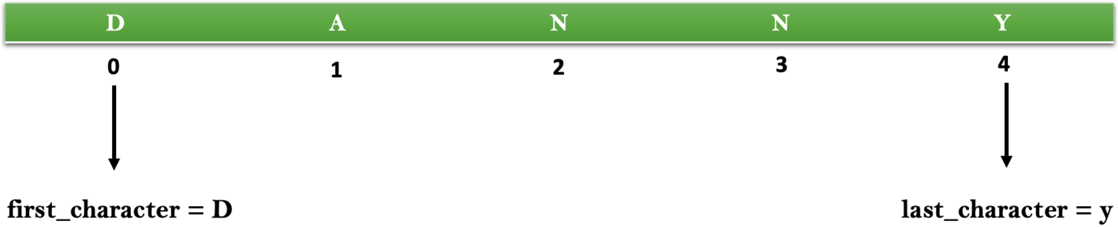
An individual character in a string is accessed using a special ***subscript*** notation in which the position is enclosed within **square brackets**. For example, suppose we want to access the character of a string **name**. Let's see an example:

```python
 #!/usr/bin/evn python
"""
A program demonstrating accessing individual character in a string
"""
name = "Danny"
first_character = name[0] # accessing first character using subscript
    notation
last_character = name[4]  # accessing last character using subscript
    notation
```

Strings and Characters

Strings & characters subscript

The previous program was able to extract two different characters from the string **"Danny"**. The first statement extracts the first character as the string **"D"** and stores it in variable **first_character**. The second statement extracts the character at position **4**, which in this case is the **last character "y"**, and stores it in a variable **last_character**.



Strings and Characters

Strings & characters subscript

Remember that all **index** value must be within the **valid range** or **length** of the character of the string. If the index value is above the length of the character positions, this would lead to an **“index out of range”** exception which will be generated at ***run time***. The len() function can be used to determine the *position of the last index*, or the last character in a string. Let’s look at an example.

```python
 #!/usr/bin/evn python
"""
A program demonstrating accessing individual character in a string
"""
name = "Danny"
# The length of the string "Danny" is 5
position = len(name) - 1 # The last position (length - 1) = 4
last_character = name[position]  # accessing last character using
    subscript notation
```

String Methods

Strings & methods

The behavior of an object is given through its methods. A **method**, juts like a function, is a *collection of programming instructions that carry out a particular task*. But unlike a function, which is a **standalone operation**, *a method can only be applied to an object of the type for which it was defined.* For example, you can apply the **upper()** method to any string, to change the characters of the string to upper case letters or characters. For a method, always note that the method name **follows the object**, and that a **dot (.)** separates the object and the method name. There is another string method called **lower()** that produces the lowercase version of a string.

String Methods

Strings & methods

It is a bit arbitrary when you need to call a function (such as **len(name)**) and when you need to call a method you use **(name.lower())**.Let's look at an example:

```
 #!/usr/bin/evn python
"""
A program illustrating string methods
"""
name = "Danny Onah"
uppercaseName = name.upper() # Sets uppercaseName to "DANNY ONAH"
print(name.lower()) # Prints danny onah
# replace the first name "Danny" with a new first name "Daniel"
name2 = name.replace("Danny", "Daniel") # Sets name2 to "Daniel Onah"
```

Important Points: String Methods

Note that:

- None of the **method calls** can change the contents of the string on which they are invoked.
- After the method call name.upper(), the name variable still holds “Danny Onah”.
- The method call only **returns the uppercase version** of the name string.
- Similarly, the **replace method** returns a new string with the replacements, **without modifying the original name string**.

Character Values

Values of a character

A character is stored internally as an **integer value**. The specific value used for a given character is based on a standard set of codes. For example, if you look up the value for the character **"H"**, you can see that it is actually encoded as the number **72**. Python provides two functions related to character encodings. The ord() function **returns the number** used to represent a given character. The chr() function returns the character associated with a given code. Let's look at an example:

```python
 #!/usr/bin/evn python
"""
A program displaying character values
"""
print("The letter H has a code of", ord("H")) # The letter H has a code
    of 72
print("Code 97 represents the character", chr(97)) # Code 97 represents
    the character a
```

Summary

Summary

Let’s revise the concepts of today’s lecture

In this lecture we discuss the following:

- We looked at **variable declaration** in Python
- We discuss about **programming with numbers, using variables and constants**
- We look at the different **arithmetic expression** in Python
- We discussed about function calling
- We discussed about Python **libraries and modules**
- We discussed about Strings, conversion and characters
- We looked at some **good programming styles and constructs**

Points to remember

Important points to take note

Let's revisit some important points:

- Put spaces around all operators (+ - * / % =, and so on) **so you expression would not be too cluttered or close together**.
- It is **customary and a convention not to put a space after a function name**. That is, write **round(x)** and not **round (x)**.
- If using parentheses for your expression ensure they are balance. Unbalanced parentheses would lead to error.
- Use **spaces** in your Python expressions and **comments** to make the code easier to read.
- **Before you can use a Python standard function, libraries and module**, you should first **import** them into your program.

Further Reading

chapters to find further reading on the concepts

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 2 Programming with Numbers & Strings*
- Learning Python (5th Edition): **By Mark Lutz** - *Chapter 5 Numeric Types, Chapter 7 String Fundamentals*
- Python Programming for absolute beginner (3rd Edition): **By Michael Dawson** - *Chapter 2 Types, Variables & Simple I/O*
- Python Crash Course - A hands-on, project-based introduction to programming (2nd Edition): **By Eric Matthes** - *Chapter 2 Variables & Simple Data Types*

Next Lecture 3

In week 3

Lecture 3: Control Flow Structures