Python Course - Introduction

Digital Career Institute

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help the learners start with basic types in Python. By the end of this submodule, the learners should be able to understand:

- How to use basic data types (print their values, know their type and know what class they are instance of).
- How to use different types of comments.
- How to use math operators.
- Basic flow in programs.
- Comparison operators.
- Conditional statements.

Topics

- **Printing out of various basic data types**
- **Comments in Python**

Digital Career Institute

Hello world in Python with print()

Digital Career Institute

- Python print() function will help you to write your very own **hello world** one-liner:

  >>> print(‘Hello world!’)

- You can use it to display formatted messages onto the screen and perhaps find some bugs.

- The simplest example of using Python print() requires just a few keystrokes:

  >>> **print()**

Hello world in Python with print()

- You don’t pass any arguments, but you still need to put empty parentheses at the end, which tell Python to actually execute the function rather than just refer to it by name.
- This will produce an invisible **newline character**, which in turn will cause a **blank line** to appear on your screen.
- A **newline character** is a special control character used to indicate the end of a line (EOL). It usually doesn’t have a visible representation on the screen, but some text editors can display such non-printable characters with little graphics.

Syntax of print() function

Digital Career Institute

```
print(object(s), sep=separator, end=end, file=file, flush=flush)
```

Parameter Values

Parameter	Description
object(s)	Any object, and as many as you like. Will be converted to string before printed
sep='*separator*'	Optional. Specify how to separate the objects, if there is more than one. Default is ' '
end='*end*'	Optional. Specify what to print at the end. Default is '\n' (line feed)
file	Optional. An object with a write method. Default is sys.stdout
flush	Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

Hello world in Python with print()

- In a more common scenario, you'd want to communicate some message to the end user. There are a few ways to achieve this.

- First, you may pass a string literal **directly** to print():

  >>> **print('Welcome to the DCI course ...')**

- **String literals** in Python can be enclosed either in single quotes (') or double quotes ("). According to the official PEP 8 style guide, you should just pick one and keep using it consistently. There's no difference, unless you need to **nest** one in another.

Nesting quotes

- What you want to do is enclose the text, which contains double quotes, within single quotes:

  'My favorite book is "Python Tricks" '

- The same trick would work the other way around:

  "My favorite book is 'Python Tricks' "

- **More about strings will be covered in another sub-module!**

Arguments of print() function

- First argument is **required**, others are **optional**.
- For example **separator** specifies how to separate the objects, if there is more than one, e.g.:

```
>>> print("Hello", "world", sep=" - ")
>>> Hello - world
```

- You can find guide about print() function in documentation!

Comments

- Comments can be used to **explain** Python code.
- Comments can be used to make the code more **readable**.
- Comments can be used to **prevent execution** when testing code.
- Comments that contradict the code are worse than no comments. **Always** make a priority of keeping the comments up-to-date when the code changes!

Block comments

- **Block comments** generally apply to some (or all) code that follows them, and are indented to **the same** level as that code.
- Each line of a block comment starts with a **#** and a **single** space (unless it is indented text inside the comment).
- Paragraphs inside a block comment are separated by a line containing a single **#**.

```
# This is a block comment

print(“Hello, World”)
```

Inline comments

- An **inline comment** is a comment on the same line as a statement.

- Inline comments should be separated by **at least two spaces** from the statement. They should start with a **#** and a **single space**.

```
print("This will run.")  # This will not run!
```

- Use inline comments **sparingly**!

Multiline comments

- Python **does not** really have a syntax for multi line comments.
- To add a multiline comment you could insert a # for **each line**:

```
# This is a comment
# written in
# more than just one line
print("Hello, World!")
```

Multiline comments

- Not quite as intended, you can use a **multiline string**.
- Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (**triple quotes**) in your code, and place your comment inside it:

```
"""
This is a comment
written in more than just one line
"""
print("Hello, World!")
```

Docstrings

- **Triple quotes** in your code are intended to use in documentation strings (a.k.a. "docstrings").
- A **docstring** is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the __doc__ special attribute of that object.
- It will be covered **later**!

At the core of the lesson

Digital Career Institute

Lesson learned:

- We know how to use print() function to print primitive data types
- We know how to use comments in Python

Variables and primitive data types

Topics

- Variables
- Primitive data types:
  - Integers (int)
  - Floats (floats)
  - Complex (complex)
  - Strings (str)
  - Booleans (bool)
- None
- Printing primitive data types

Digital Career Institute

Variables

- **Variables** are containers for storing data values.
- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Variables - examples

- x = 5
- my_name = "Joe"
- number = 123
- Variables in Python do not need to be declared with any particular ***type***:
  - x = 4 # x is of type **int**
  - x = "Sally" # x is now of type **str**

Variables - casting

- If you want to specify the data type of a variable, this can be done with **casting**.
- `x = str(31)  # x will be '31'`
- `y = int(3)  # y will be 31`
- `z = float(3)  # z will be 31.0`

Variables - case sensitivity

- Variable names are **case-sensitive**.
- c = 42
- C = "John"
- # C will not overwrite c

Variable names

- A variable name **must** start with a letter or the underscore character.
- A variable name **cannot** start with a number.
- A variable name can only contain **alphanumeric** characters and underscores (A-z, 0-9, and _).
- Variable names are **case-sensitive** (age, Age and AGE are three different variables).

Multi word variable names

- Variable names with more than one word can be **difficult** to read.
- There are several **techniques** you can use to make them more readable:
  - **Camel Case**
  - Each word, except the first, starts with a capital letter:
  - myVariableName = "John"

Multi word variable names

- Variable names with more than one word can be **difficult** to read.
- There are several **techniques** you can use to make them more readable:
  - **Pascal Case**
  - Each word starts with a capital letter:
  - MyVariableName = "John"

Multi word variable names

- Variable names with more than one word can be **difficult** to read.
- There are several **techniques** you can use to make them more readable:
  - **Snake Case**
  - Each word is separated by an underscore character:
  - my_variable_name = "John"

Assign multiple values

- Python allows you to assign values to multiple variables in one line:
  - x, y, z = "Red", "Green", "Blue"
  - Now:
    - x = “Red”
    - y = “Green”
    - z = “Blue”

- Make sure the number of variables matches the number of values, or else you will get an **error**!

Assign one value to multiple variables

Digital Career Institute

- You can assign the ***same*** value to **multiple variables** in one line:
  - x = y = z = 345
  - Now:
    - x = 345
    - y = 345
    - z = 345

Swapping variables

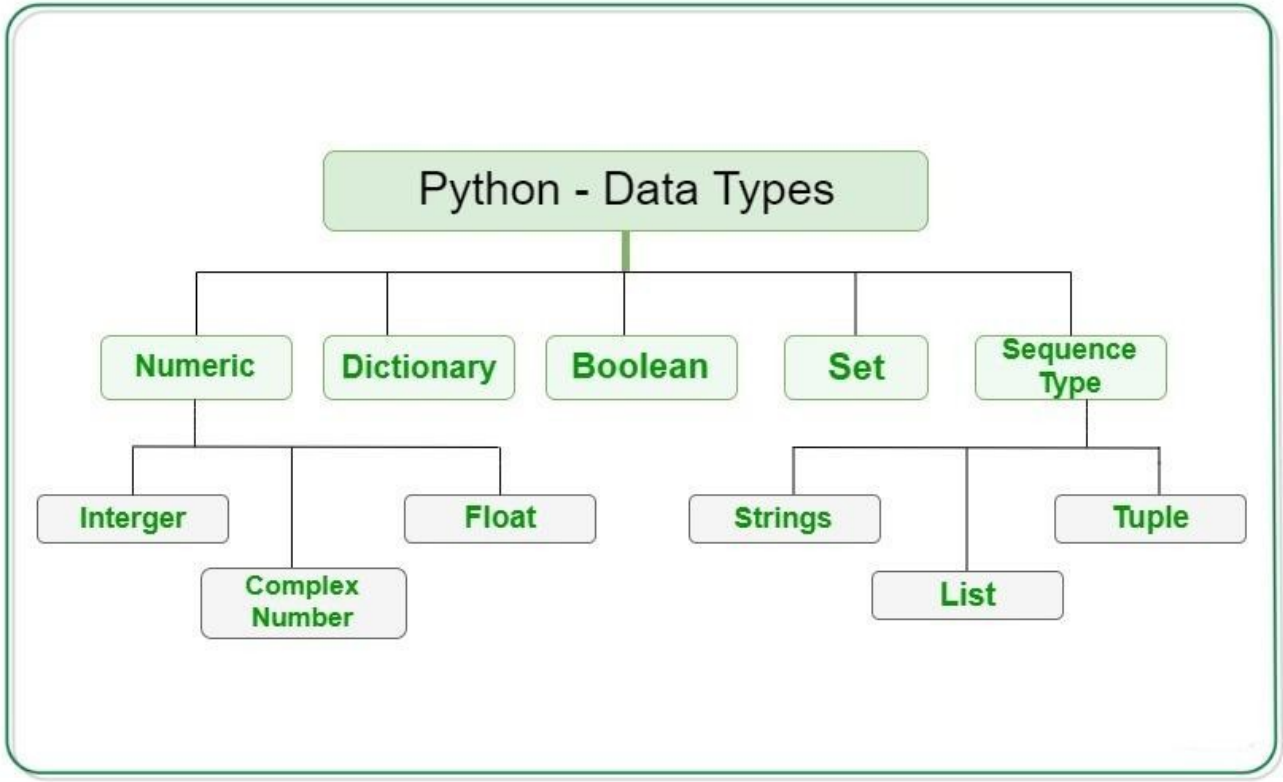
- In Python, there is a simple construct to **swap** variables:

```
>>> x = 5
>>> y = 10
>>> x, y = y, x
>>> x
10
>>> y
5
```

Type hints

- Type hints will be covered later, as they are useful while working with functions in Python!

Python data types



Useful function - type()

- Type objects represent the various object types. An object's **type** is accessed by the built-in function type(). There are no special operations on types.
- With one argument, return the type of an *object*.
- Types are written like this: **<class 'int'>**.
- For example, in shell **type("Hello")** returns – **<class 'str'>**

Useful function - isinstance()

Digital Career Institute

Syntax

```
isinstance(object, type)
```

Parameter Values

Parameter	Description
object	Required. An object.
type	A type or a class, or a tuple of types and/or classes

isinstance() - examples

Digital Career Institute

```
>>> isinstance(123, int)
True
>>> isinstance("hello", float)
False
>>> isinstance(123.456, float)
True
>>> isinstance(4+2j, complex)
True
>>> isinstance(False, bool)
True
>>> isinstance("hi", str)
True
```

Integers

- Python interprets a sequence of decimal digits without any prefix to be a **decimal** number.

- In Python 3, there is effectively no limit to how long an **integer** value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be **as long as you need** it to be.

Integers

- I typed command below in my Python shell and **it works** 😀

- print(12312312312312312312312312312312312312312312323233242345234342342342342342342342343747384757384758237483784723848237487238478378747334434545343434690394399292438482848929284446736476563576375635855674856454548758758475874587458478574857847584758748578457845785748577 + 1)

- You can check very big integers in your shell!

Floating-point numbers

- The **float** type in Python designates a floating-point number.
- **float** values are specified with a decimal point (e.g. **4.34**).
- Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation (e.g. **4e3**)

```
>>> type(.23)
<class 'float'>
>>> 4e3
4000.0
>>> type(4e3)
<class 'float'>
```

Complex numbers

- Complex numbers are specified as **<real part>+<imaginary part>j**. For example: 3 + 5j

- Complex number literals in Python mimic the mathematical notation, which is also known as the **standard form**, the **algebraic form**, or sometimes the **canonical form**, of a complex number.

- In Python, you can use either lowercase j or uppercase J in those literals.

Complex numbers - example

```
>>> 4-5j
(4-5j)
>>> type(4-5j)
<class 'complex'>
>>> type(3+2.3j)
<class 'complex'>
>>> 1+j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> 1+1j
(1+1j)
```

- Imaginary part of complex number can’t be only **j** or **J** letter, you must type number before it!

Strings

- Strings are sequences of character data. The string type in Python is called str.
- String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string.
- They will be covered in separate sub-module.

Boolean

- Python 3 provides a Boolean data type.
- Objects of **Boolean** type may have one of two values, **True** or **False**.
- In programming you often need to know if an expression is **True** or **False**.
- You can evaluate any expression in Python, and get one of these two answers

Boolean - examples

- You can evaluate expression or you can use built-in bool() function.

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(None)
False
>>> bool(True)
True
>>> bool(0+0j)
False
```

```
>>> bool("")
False
>>> 10 > 9
True
>>> 10 == 9
False
>>> bool("")
False
>>> bool("text")
True
```

Boolean - most values are True

- Almost any value is evaluated to **True** if it has some sort of content.
- Any string is **True**, **except** empty strings.
- Any number is **True**, **except** 0.
- Any list, tuple, set, and dictionary are **True**, **except** empty ones (all these data types will be covered later).

Boolean - some values are False

- In fact, there are not many values that evaluate to False, except empty values, such as:
  - () - empty set
  - [] - empty list
  - {} - empty dictionary
  - "" - empty string
  - the number 0,
  - the value **None**,
  - and of course the value **False** evaluates to False

None

- In some languages, variables come to life from a **declaration**. They don’t have to have an initial value assigned to them. In those languages, the initial default value for some types of variables might be **null**.

- In Python, however, variables come to life from **assignment statements**.

- The **None** keyword is used to define a **null** value, or **no value** at all.

None

- None is **not the same** as 0, False, or an empty string.
- None is a data type of its own (**NoneType**) and only None can be None.
- We can assign **None** to any variable, but you **can not** create other NoneType objects.
- Some usages of None will be covered later!

None

```
>>> print(bar)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
>>> bar = None
>>> print(bar)
None
```

Printing primitive data types

Digital Career Institute

- None is **not the same** as 0, False, or an empty string.
- None is a data type of its own (**NoneType**) and only None can be None.
- We can assign **None** to any variable, but you **can not** create other NoneType objects.
- Some usages of None will be covered later!

Printing primitive data types

Digital Career Institute

```
>>> print(42)
42
>>> print(3.14)
3.14
>>> print(True)
True
>>> print(1 + 2j)
(1+2j)
>>> print('Hello')
Hello
>>> print(None)
None
```

At the core of the lesson

Lesson learned:

- We know how to work with variables in Python
- We know how to handle the primitive data types in Python

Operators and basic math functions

Topics

- Math operators
- Basic math functions
- Assignment operators

Digital Career Institute

Operators and operands

- **Operators** are special symbols that represent computations like addition and multiplication.
- The values the operator uses are called **operands**.

Expression

- An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:
- **20+32 hour-1 hour*60+minute minute/60 5**2 (5+9)*(15-7)**

Python math operators

- Arithmetic operators are used with numeric values to perform common mathematical operations:
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Modulus
  - Exponentiation
  - Floor division

Python math operators

Digital Career Institute

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python math operators - examples

Digital Career Institute

```
>>> 2 + 3
5
>>> 10 - 4
6
>>> 3 * 4
12
>>> 12 / 2
6.0
>>> 12 / 5
2.4
```

```
>>> 10 % 2
0
>>> 10 % 3
1
>>> 2 ** 3
8
>>> 10 // 3
3
>>> 10 // 4
2
```

Python math operators - examples

- Modulus % returns the rest from division, for example:
  - 10 % 3 = 1, because 10 ÷ 3 = 3 + 1 (remainder) or 3 * 3 + 1 = 10

- Floor division returns integer part of the result, for example:
  - 10 // 4 = 2, because 10 ÷ 4 = 2.5 and the integer part of 2.5 is just number 2.

Built-in math functions

min() and max()

- The min() and max() functions can be used to find the lowest or highest value in a set of values:

```
>>> x = max(5, 10, 15)
>>> print("Max. value is", x)
Max. value is 15
>>> y = min(5, 10, 15)
>>> print("Min. value is", y)
Min. value is 5
```

abs()

- The abs() function returns the **absolute** (positive) value of the specified number:

```
>>> abs(12.34)
12.34
>>> abs(-12.34)
12.34
>>> abs(3 + 4j)
5.0
>>> abs(True)
1
>>> abs(False)
0
```

pow()

- The pow(*x*, *y*) function returns the value of x to the **power** of y (x^y):

```
>>> pow(2, 3)
8
>>> pow(2, 4)
16
>>> pow(2, 5)
32
>>> pow(-2, 3)
-8
>>> pow(2.5, 2)
6.25
```

round()

- The round() function returns a floating point number that is a **rounded version** of the specified number, with the specified number of decimals.

- The default number of decimals is 0, meaning that the function **without** second argument will return the **nearest** integer:

```
>>> round(1.23)
1
>>> round(1.56)
2
```

round()

Syntax

round(*number, digits*)

Parameter Values

Parameter	Description
number	Required. The number to be rounded
digits	Optional. The number of decimals to use when rounding the number. Default is 0

round() with second argument

Digital Career Institute

```
>>> round(1.23456, 2)
1.23
>>> round(1.23456, 4)
1.2346
```

math module

- Python has also a built-in module called **math**, which extends the list of mathematical functions.
- To use it, you must **import** the math module:

```
>>> import math
```

- Importing and modules will be covered **later** in detail!

math.sqrt()

- When you have imported the math module, you can start using methods and constants of the module.

- The math.sqrt() method for example, returns the square root of a number:

```
>>> w = math.sqrt(2)
>>> s = math.sqrt(49)
>>> print("Square root of 2 is", w)
Square root of 2 is 1.4142135623730951
>>> print("Square root of 49 is", s)
Square root of 49 is 7.0
```

math.ceil() and math.floor()

- The math.**ceil()** method rounds a number **upwards** to its nearest integer, and the math.**floor()** method rounds a number **downwards** to its nearest integer, and returns the result:

```
>>> c = math.ceil(3.14)
>>> print(c)
4
>>> f = math.floor(3.14)
>>> print(f)
3
```

Assignment operators

Python assignment operators

Digital Career Institute

- Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

Python assignment operators

- Examples:

```
>>> x = 3
>>> y = 5
>>> x += 3
>>> print("x =", x)
x = 6
>>> y += 4
>>> print("y =", y)
y = 9
```

Number systems

- There are an infinite number of ways to represent numbers. Most modern civilizations use positional notation, which is efficient, flexible, and well suited for doing arithmetic.
- A notable feature of any positional system is its **base**, which represents the number of digits available. People naturally favor the **base-ten** numeral system, also known as the **decimal system**, because it plays nicely with counting on fingers.

Number systems

- Computers, on the other hand, treat data as a bunch of numbers expressed in the **base-two** numeral system, more commonly known as the **binary** system. Such numbers are composed of only two digits, zero and one.

- For example, the binary number 10011100_2 is equivalent to 156_{10} in the base-ten system. Because there are ten numerals in the decimal system (zero through nine) it usually takes fewer digits to write the same number in base ten than in base two.

What is base in numeral systems?

- A notable feature of any positional system is its **base**, which represents the number of digits available.

- People naturally favor the **base-ten** numeral system, also known as the **decimal system** (we have 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 in this system - 10 digits) , because it plays nicely with counting on fingers.

- Computers, on the other hand, treat data as a bunch of numbers expressed in the **base-two** numeral system, more commonly known as the **binary** system. Such numbers are composed of only **two** digits - zero and one.

Octal and hexadecimal systems

- The **octal** numeral system, or **oct** for short, is the base-8 number system, and uses the digits 0 to 7,

- In mathematics and computing, the **hexadecimal** (also **base 16** or **hex**) numeral system is a positional numeral system that represents numbers using a radix (base) of 16.

- Unlike the common way of representing numbers using 10 symbols, hexadecimal uses **16** distinct symbols, most often the symbols "0" – "9" to represent values **0 to 9**, and "A" – "F" (or alternatively "a" – "f") to represent values **10 to 15**.

Integers with base other than 10

Digital Career Institute

Prefix	Interpretation	Base
`0b` (zero + lowercase letter `'b'`) `0B` (zero + uppercase letter `'B'`)	Binary	2
`0o` (zero + lowercase letter `'o'`) `0O` (zero + uppercase letter `'O'`)	Octal	8
`0x` (zero + lowercase letter `'x'`) `0X` (zero + uppercase letter `'X'`)	Hexadecimal	16

Integers with base other than 10

- Binary numeral system
- Octal numeral system
- Hexadecimal numeral system

- Just use the right prefix!

```
>>> print(0b10)
2
>>> print(0o10)
8
>>> print(0x10)
16
```

Binary numbers

- **Binary number** is a number expressed in the base-2 numeral system or binary numeral system, a method of mathematical expression which uses **only two** symbols: typically "0" (zero) and "1" (one).
- How to count in binary?

Binary		
0		We start at 0
1		Then 1
???		But then there is no symbol for 2 ... what do we do?

Binary counting

Well how do we count in Decimal?	
0	Start at 0
...	Count 1, 2, 3, 4, 5, 6, 7, 8, and then...
9	This is the **last digit** in Decimal
10	So we start back at 0 again, but add 1 on the left

Binary counting

The same thing is done in binary:		
	0	Start at 0
.	1	Then 1
..	10	Now start back at 0 again, but **add 1 on the left**
...	11	1 more
....	???	But NOW what … ?

Binary vs Decimal

Decimal vs Binary

Here are some **equivalent** values:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Binary numbers

- When you **say** a binary number, pronounce each digit (example, the binary number "101" is spoken as "*one zero one*", or sometimes "*one-oh-one*"). This way people don't get confused with the decimal number.
- Detailed explanation of binary number system: here
- Binary numbers on Wikipedia

Integer to binary

- Built-in function **bin()** converts an integer number to a binary string prefixed with “0b” (zero and b). The result is a valid Python expression:

```
>>> bin(7)
'0b111'
>>> bin(8)
'0b1000'
>>> bin(1000)
'0b1111101000'
```

Binary to integer

- Built-in function **int()** returns an integer object constructed from a number or string.

- To convert from binary to integer we must set second argument of int() called **base**. For binary numbers the base is **2**:

```
>>> int('111', base=2)
7
>>> int('111', 2)
7
>>> int('101010101', 2)
341
```

Bitwise operators

- Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Bitwise AND

Expression	Binary Value	Decimal Value
a	10011100_2	156_{10}
b	110100_2	52_{10}
a & b	10100_2	20_{10}

```
>>> 156 & 52
20
```

Bitwise OR

Expression	Binary Value	Decimal Value
a	10011100_2	156_{10}
b	110100_2	52_{10}
a | b	10111100_2	188_{10}

```
>>> 156 | 52
188
```

Bitwise shift operator

- Bitwise shift operators are another kind of tool for **bit manipulation**.
- They let you move the bits around, which will be handy for creating **bitmasks** later on. In the past, they were often used to improve the speed of certain mathematical operations.
- We will explore left and right shifting

Left shift operator

- The bitwise **left** shift operator (**<<**) moves the bits of its **first** operand to the left by the number of places specified in its **second** operand. It also takes care of inserting enough zero bits to fill the gap that arises on the right edge of the new bit pattern.

- Shifting a single bit to the left by o**ne place doubles** its value. For example, instead of a two, the bit will indicate a four after the shift. Moving it two places to the left will quadruple the resulting value. When you add up all the bits in a given number, you'll notice that it also gets doubled with every place shifted:

Left shift operator

Digital Career Institute

Expression	Binary Value	Decimal Value
`a`	100111_2	39_{10}
`a << 1`	1001110_2	78_{10}
`a << 2`	10011100_2	156_{10}
`a << 3`	100111000_2	312_{10}

Right shift operator

- The bitwise **right** shift operator (**>>**) is analogous to the left one, but instead of moving bits to the left, it pushes them to the right by the specified number of places. The rightmost bits always get dropped.

- Every time you shift a bit to the right by one position, you **halve** its underlying value. Moving the same bit by two places to the right produces a quarter of the original value, and so on. When you add up all the individual bits, you'll see that the same rule applies to the number they represent:

Right shift operator

Digital Career Institute

Expression	Binary Value	Decimal Value
`a`	10011101_2	157_{10}
`a >> 1`	1001110_2	78_{10}
`a >> 2`	100111_2	39_{10}
`a >> 3`	10011_2	19_{10}

At the core of the lesson

Digital Career Institute

Lesson learned:

- We know math operators and how to use them
- We know basic math functions and how to use them
- We know assignment operators and how to use them
- We know basic binary operators and how to use them

Basic control flow

Topics

- Basic control flow
- If statement
- Repetition statements
- Comparison operators
- Pass statement

Digital Career Institute

What is control flow?

- A program’s **control flow** is the order in which the program’s code executes.
- The control flow of a Python program is regulated by **conditional statements**, **loops**, and **function calls**.

Control structures

Python has *three* types of control structures:

- **Sequential** - default mode.
- **Selection** - used for decisions and branching.
- **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

Sequential statements

- **Sequential statements** are a set of statements whose execution process happens in a sequence.

- The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

Sequential statement - example

Digital Career Institute

```
1  # This is a sequential statement
2
3  a = 20
4  b = 10
5  c = a - b
6  print("Result of subtraction is : ", c)
```

Selection/decision control statements

- The **selection statement** allows a program to test several conditions and execute instructions based on which condition is **true**.

- In Python, the selection statements are also known as **decision control statements** or **branching statements**.

Selection/decision control statements

Digital Career Institute

Some Decision Control Statements are:

- Simple **if**
- **If-else**
- nested **if**
- **if-elif-else**

If statements

- ***If statements*** are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied.

- A *simple* ***if*** only has one condition to check.

Indentation

- Python relies on indentation (whitespace at the beginning of a line) to define scope in the code.
- Preferred indentation is equal to **4 spaces**
- Other programming languages often use curly-brackets for this purpose.

Indentation

- If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

- The **elif** keyword is pythons way of saying "if the previous conditions were not true, then try this condition":

```
a = 33

b = 33

if b > a:

        print("b is greater than a")  # indentation

elif a == b:

            print("a and b are equal")  # indentation
```

Else

- The **else** keyword catches anything which isn't caught by the preceding conditions:

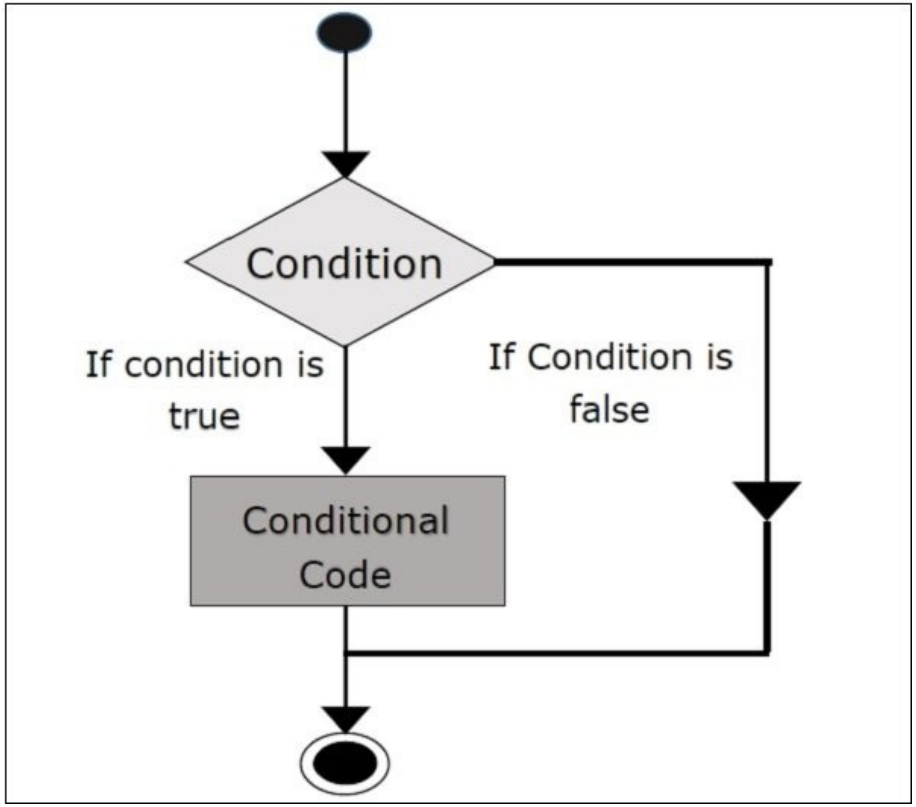
```
a = 200
b = 33
if b > a:
        print("b is greater than a")
elif a == b:
        print("a and b are equal")
else:
                print("a is greater than b")
```

“Short” if

- If you have **only one** statement to execute, you can put it on the same line as the if statement:

```
if a > b: print("a is greater than b")
```

Simple **if**


Condition
If condition is true
If Condition is false
Conditional Code

Simple if - example

```
1  n = 10
2  if n % 2 == 0:
3      print("n is an even number!")
```

If -else

- The **if-else** *statement* evaluates the condition and will execute the body of **if**, if the test condition is True.

- But if the condition is **False**, then the body of else is executed.

If -else (algorithm)

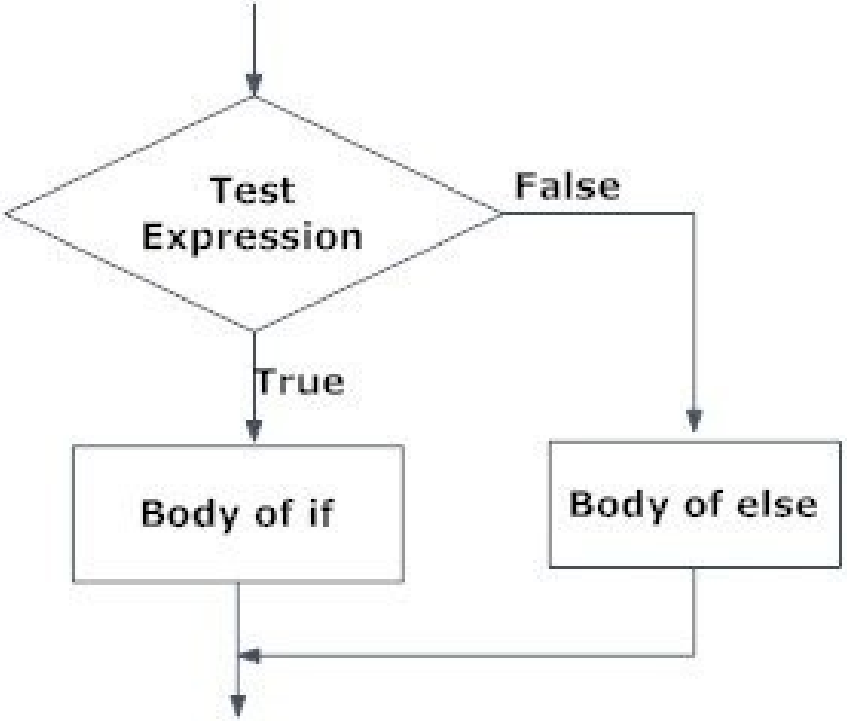
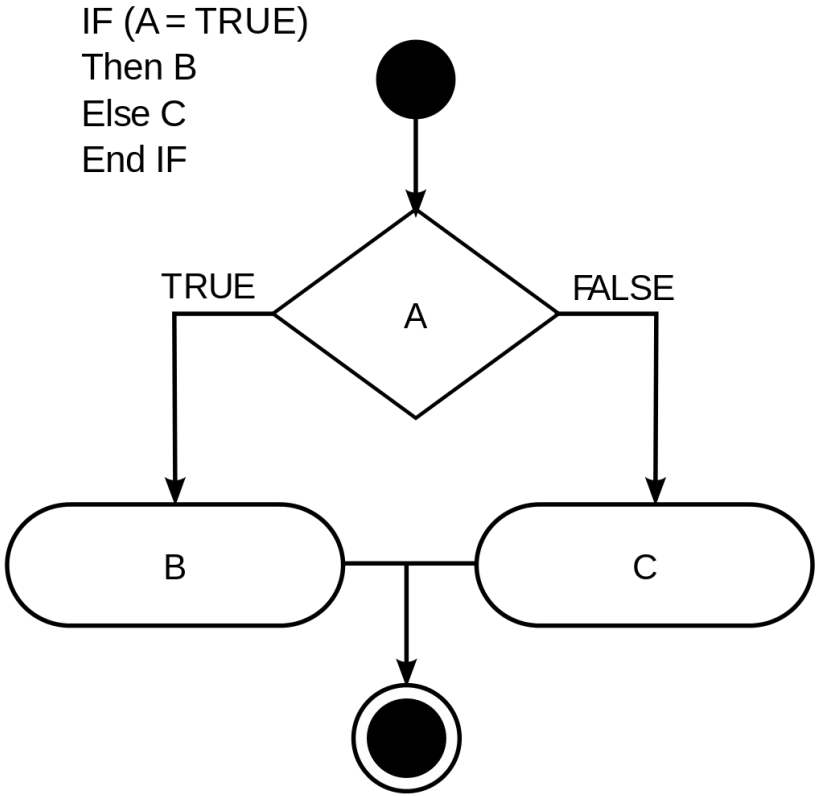


Fig: Operation of if...else statement

If -else (algorithm ver. 2)



If -else (code example)

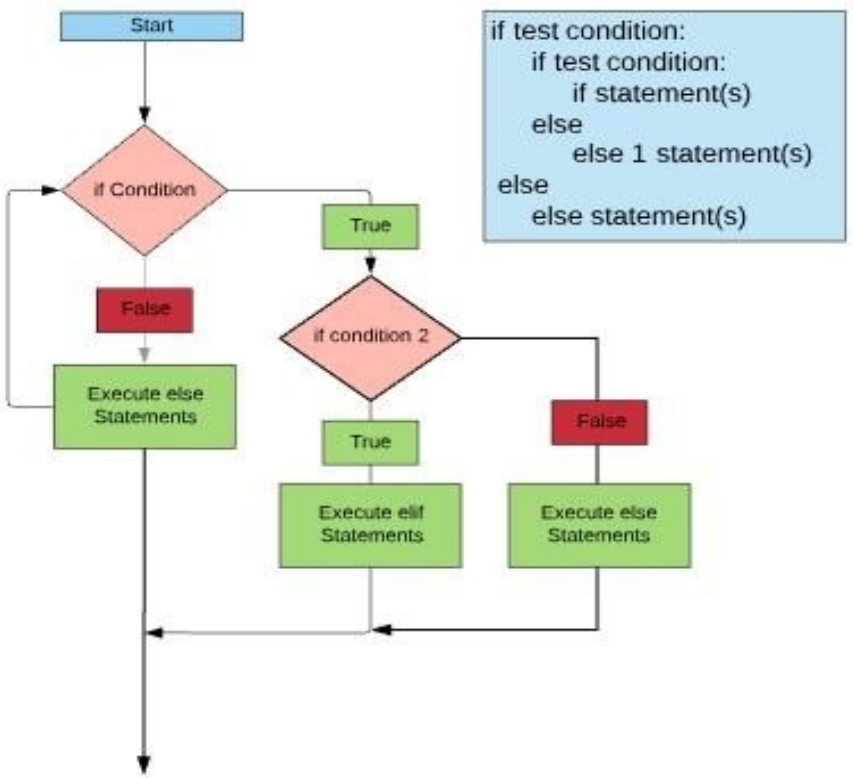
Digital Career Institute

```
n = 5
if n % 2 == 0:
    print("n is even!")
else:
    print("n is odd!")
```

Nested If

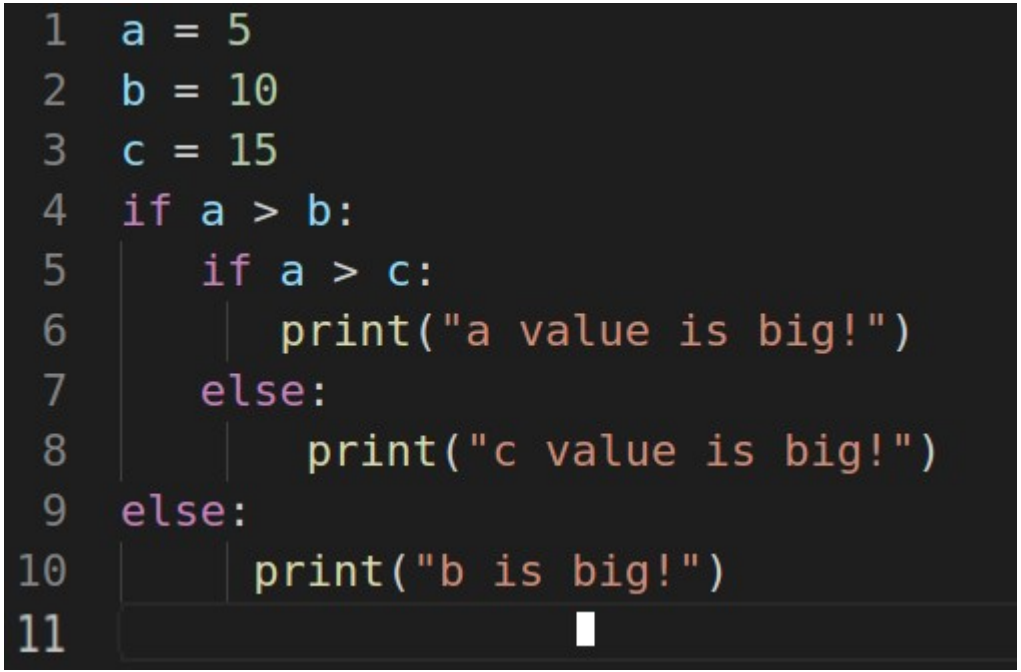
- **Nested** if st*atements* are an if statement **inside** another **if** statement.

Nested If




```
if test condition:
    if test condition:
        if statement(s)
    else
        else 1 statement(s)
else
    else statement(s)
```

Nested If


1 a = 5
2 b = 10
3 c = 15
4 if a > b:
5 if a > c:
6 print("a value is big!")
7 else:
8 print("c value is big!")
9 else:
10 print("b is big!")
11

If-elif-else

- **if-elif-else:** The *if-elif-else statement* is used to conditionally execute a statement or a block of statements.

If-elif-else

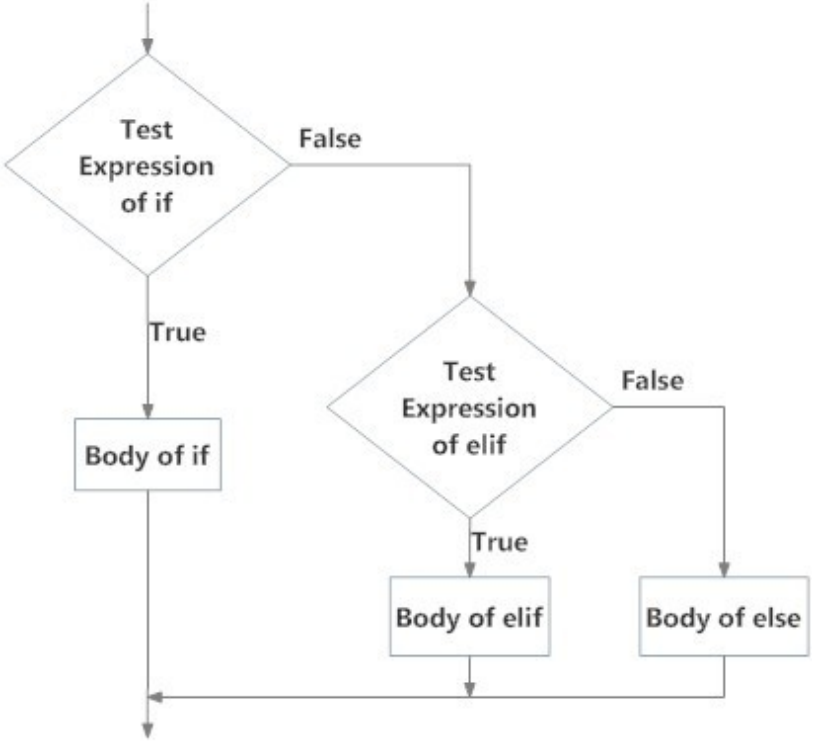


Fig: Operation of if...elif...else statement

If-elif-else

Digital Career Institute

```
1  x = 15
2  y = 12
3
4  if x == y:
5      print("Both are equal!")
6  elif x > y:
7      print("x is greater than y")
8  else:
9      print("x is smaller than y")
10
```

Python comparison operators

- Comparison operators are used to **compare** two values.
- The result is always **True** or **False**!

Python comparison operators

Digital Career Institute

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Python comparison operators - example

Digital Career Institute

```
x = 5
y = 3

print(x > y)

# returns True because 5 is greater than 3
```

Repetition statements

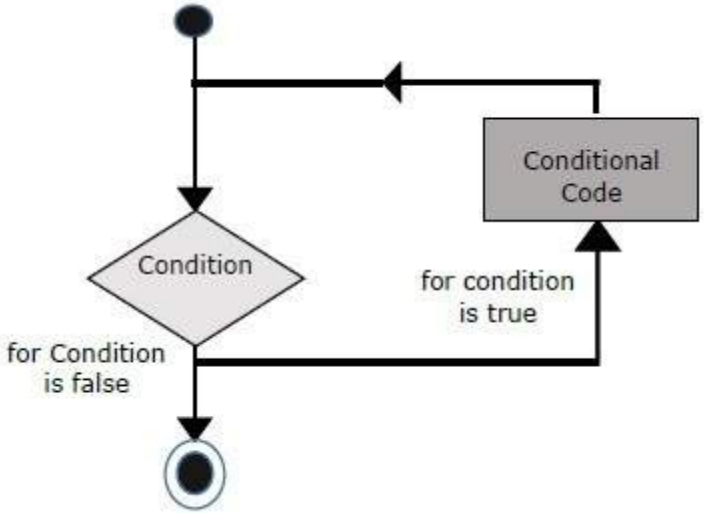
Repetition statement

- A **repetition statement** is used to repeat a group (block) of programming instructions.
- In Python, we generally have two loops/repetitive statements:
  - **for** loop
  - **while** loop
- A **for** and **while** loops will be covered later in more detail!

For loop

- A **for loop** is used to iterate over a sequence that is either a list, tuple, dictionary, or a set (will be covered **later**!).
- We can execute a set of statements once for each item in a sentence.

For loop


Conditional
Code
Condition
for condition
is true
for Condition
is false

range() function

- To loop through a set of code a specified number of times, we can use the range() function,

- With **one** argument, e.g. **range(5)**, function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number (but without this number)

```
for x in range(5):
    print(x)
# prints integers 0, 1, 2, 3, 4
# without number 5
```

range() function

- It is possible to specify the starting value by adding a parameter: **range(2, 6)**, which means values from 2 to 6 (but not including 6):

```
for x in range(2, 6):
    print(x)
# prints integers 2, 3, 4, 5
# without number 6
```

range() function

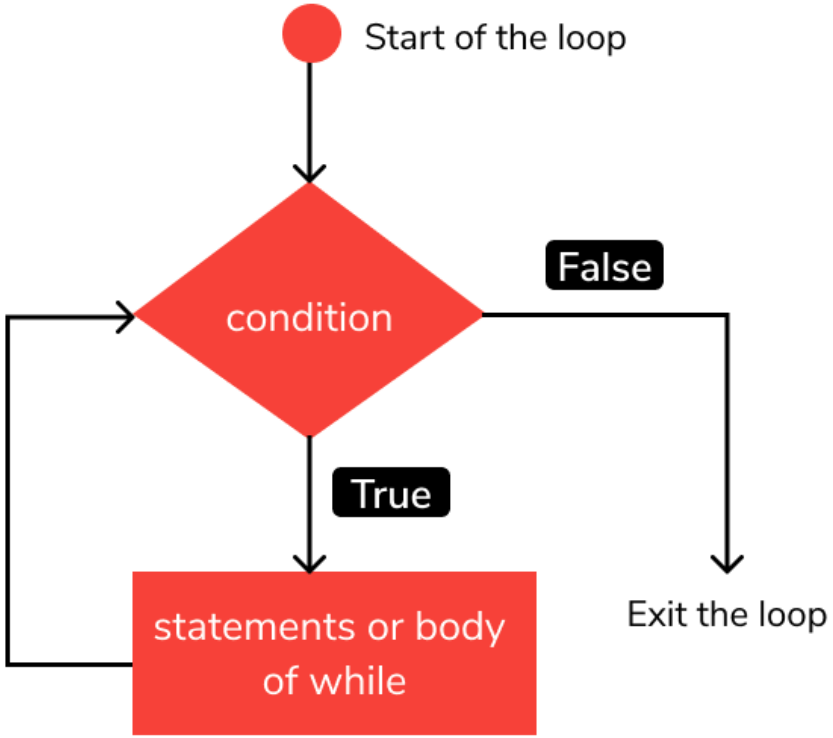
- The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a **third** parameter:

```
for x in range(2, 15, 3):
    print(x)
# prints integers 2, 5, 8, 11, 14
```

While loop

- In Python, **while loops** are used to execute a block of statements repeatedly until a given condition is satisfied.
- Then, the expression is checked again and, if it is **still true**, the body is executed again.
- This continues until the expression becomes false.

While loop


Start of the loop
condition
False
True
statements or body of while
Exit the loop

While loop

```
i = 1
while i < 6:
    print(i)
    i += 1
# prints 1, 2, 3, 4, 5
```

- Remember to increment i, or else the loop will continue **forever**.
- The while loop requires relevant variables to be ready, in this example we need to define an indexing variable **i**, which we set to 1.

else in **for** loop

- The **else** keyword in a **for** loop specifies a block of code to be executed when the loop is finished:

```
for x in range(3):
    print(x)
else:
    print("Finally finished!")
# prints 0, 1, 2, "Finally finished!"
```

Nested loops

- A nested loop is a loop inside a loop.
- The **"inner loop"** will be executed one time for each iteration of the "**outer loop**":

```
for x in range(10, 21, 10):  # x will be 10, and then 20 in second iteration
    for y in range(2):       # y will 0, and then 1 in second iteration
        print(x + y)
# prints 10, 11, 20, 21
```

Pass statement

- The **pass** statement is used as a placeholder for future code.
- When the **pass** statement is executed, nothing happens, but you avoid getting an **error** when empty code is not allowed.
- Empty code **is not allowed** in loops, function definitions, class definitions, or in if statements.

Pass statement

- for loops **cannot** be empty, but if you for some reason have a for loop with no content, put in the **pass** statement to avoid getting an error.

```
for x in range(10):
    pass
# # having an empty for loop like this,
# would raise an error without the pass statement
```

At the core of the lesson

Lesson learned:

- We know basic control flow rules
- We know comparison operators
- We know repetition statements

Digital Career Institute

Additional operators

Topics

- Arithmetic operations
- Logical operators
- Ternary operator
- Conditional statements in if statement

Digital Career Institute

Logical operators

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Ternary operator (short if-else)

- If you have only one statement to execute, one for **if**, and one for **else**, you can put it all on the same line:

```
a = 2

b = 330

print("A") if a > b else print("B")
```

Identity operators

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

- It will be covered **later**!

Membership operators

- Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

- It will be covered **later**!

At the core of the lesson

Digital Career Institute

Lesson learned:

- We know comparison operators
- We know logical operators
- We know the idea of identity and membership operators

Command Line Interface

Topics

- Read parameters in CLI context
- input() function
- cmd
- sys
- getopt

Digital Career Institute

Command Line Interface (CLI)

- CLI provides a way for a user to **interact** with a program running in a text-based shell interpreter.
- Some examples of shell interpreters are Bash on Linux or Command Prompt on Windows.
- A command line interface is enabled by the shell interpreter that exposes a command prompt.

Command prompt

- A **command prompt** (or just *prompt*) is a sequence of (one or more) characters used in a command-line interface to indicate **readiness** to accept commands.
- It literally prompts the user to take action.
- A prompt usually ends with one of the characters $, %, #, : , > or - and often includes other information, such as the path of the current working directory and the hostname.

Command prompt

- **Command prompt** can be characterized by the following elements:
  - A **command** or program
  - Zero or more command line **arguments**
  - An **output** representing the result of the command
  - Textual documentation referred to as **usage** or **help**
  - Not every command line interface may provide all these

Command prompt - example no. 1

- In this example, the Python interpreter takes option -c for **command**, which says to execute the Python command line arguments following the option -c as a Python program.

```
artur@artur-MSI:~$ python3 -c "print('Welcome to DCI course')"
Welcome to DCI course
```

Command prompt - example no. 1

- This example shows how to invoke Python with -h to display the help:

```
artur@artur-MSI:~$ python3 -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
```

Command Line Arguments

- The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**. Python provides various ways of dealing with these types of arguments. The three most common are:
  - sys.argv
  - getopt module
  - argparse module

Python Command Line Arguments

Digital Career Institute

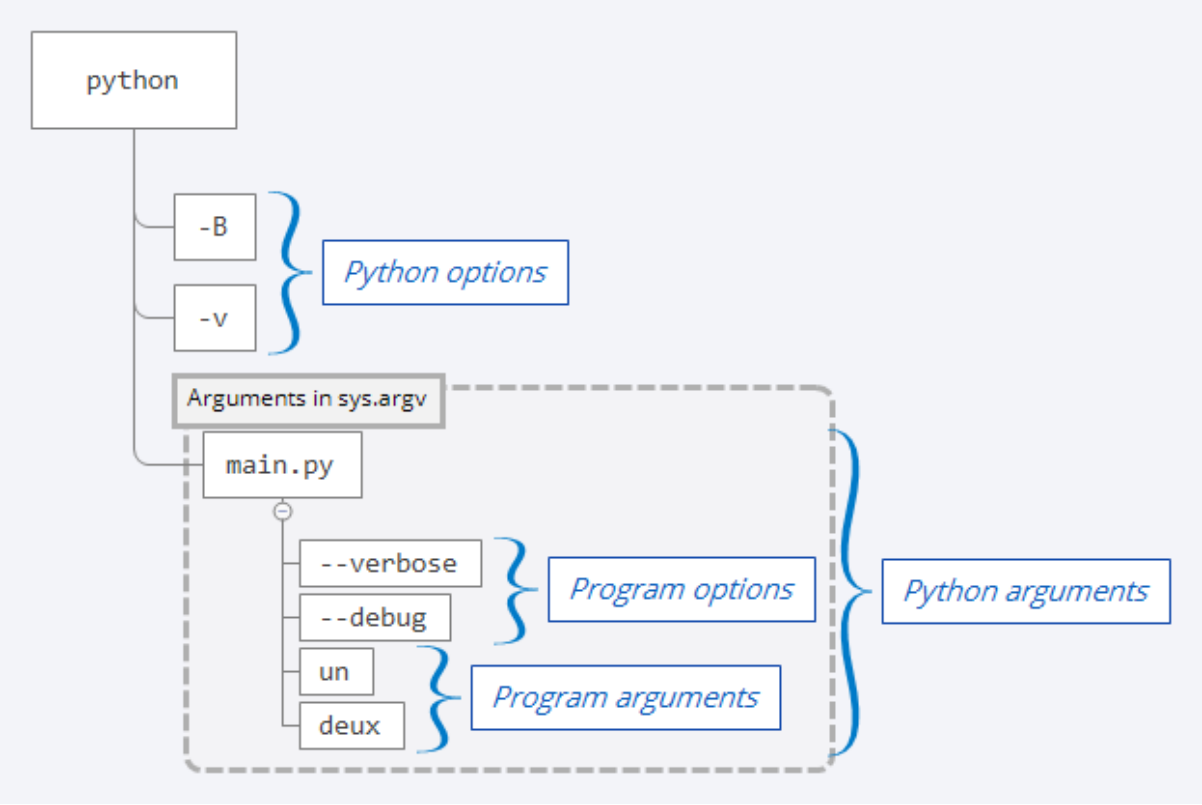
Python command line arguments are a subset of the command line interface. They can be composed of different types of arguments:

1. **Options** modify the behavior of a particular command or program.

2. **Arguments** represent the source or destination to be processed.

3. **Subcommands** allow a program to define more than one command with the respective set of options and arguments.

Python Command Line Arguments



sys module

- The **sys module** in Python provides various functions and variables that are used to manipulate different parts of the Python runtime environment.

- It allows operating on the interpreter as it provides access to the variables and functions that interact strongly with the interpreter.

sys.argv

- **argv** is a variable provided by the **sys** module which holds a list of all the arguments passed to the command line (including the script name).

- So even if you don’t pass any arguments to your script. The argv variable always contains **at least one** element i.e the script name.

- The arguments in argv are always parsed as **string**. So be careful if you are expecting your input to be of any other type. You may need to cast or convert the elements according to your requirements.

sys.argv

- The sys module exposes an array named **argv** that includes the following:
  - **argv[0]** contains the name of the current Python program.
  - **argv[1:]**, the rest of the list, contains any and all Python command line arguments passed to the program.

sys.argv - example

```
1 # argv.py
2 import sys
3
4 print(f"Name of the script      : {sys.argv[0]=}")
5 print(f"Arguments of the script : {sys.argv[1:]=}")
```

- **Line 2** imports the internal Python module sys.
- **Line 4** extracts the name of the program by accessing the first element of the list sys.argv.
- **Line 5** displays the Python command line arguments by fetching all the remaining elements of the list sys.argv.

sys.argv - example

- After execution of code in file argv.py :

```
artur@artur-MSI:~/Desktop/DCI$ python3 argv.py some arguments here
Name of the script      : sys.argv[0]='argv.py'
Arguments of the script : sys.argv[1:]=['some', 'arguments', 'here']
```

sys.argv - summing arguments (code)

Digital Career Institute

```python
import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("Name of Python script:", sys.argv[0])

# Addition of numbers
sum_of_arguments = 0

for i in range(1, n):
    sum_of_arguments += int(sys.argv[i])

print("Result:", sum_of_arguments)
```

sys.argv - summing arguments (results)

Digital Career Institute

```
artur@artur-MSI:~/Desktop/DCI$ python3 argv-summing.py 1 2 3 4
Total arguments passed: 5
Name of Python script: argv-summing.py
Result: 10
```

sys.argv

- sys.argv is of the type **<list>** so you can access the elements just as you would from any other list. For example, **sys.argv[1]**

- It does not provide any inherent mechanism to make any of the arguments as required or optional and we also cannot limit the number of arguments supplied to our script.

- Sys.argv can be more than sufficient if your problem definition is simple enough. But if your requirements are a bit more advanced than just adding two numbers, you may need to use **getopt** or **argparse**.

getopt module

- **getopt** provides us with features that make it **easier** to process command line arguments in Python.
- **getopt** is a module that comes bundled with any standard python installation and therefore you **need not** install it explicitly.
- A major advantage of getopt over just using sys.argv is getopt supports **switch style options** (for example: -s or --sum).
- Hence getopt supported options are **position-independent**. The example $ ls -li works the same as $ ls -il

getopt module

- The options are of two types:
  - Options that need a value to be passed with them. These are defined by the option name suffixed with **=** (for example: **num1=**)
  - Options that behave as a flag and do not need a value. These are defined by passing the option name **without the suffix =** (for example: --subtract)

getopt module

- The options can have two variations:
  - **shortopts** are one letter options, denoted by prefixing a single - to an option name (for example, $ ls -l)
  - **longopts** are a more descriptive representation of an option, denoted by prefixing two – to an option name (for example, $ ls --long-list)

getopt (usage)

- getopt module provides a **getopt**(args, shortopts, longopts=[]) function which we can use to define our options:

- Code of getopt() **function** usage:

```
(opts, args) = getopt.getopt(sys.argv[1:], 'ha:b:s', ['help','num1=', 'num2=', 'subtract'])
```

- sys.argv holds the unformatted list of all the arguments passed to a python script.

getopt (usage)

- There are two variables used (**opts**, **args**) because getopt.getopt function returns two elements:
  - one containing a <list> of **options**
  - second has a <list> of **arguments** that are not specified in our getopt initialization.

getopt (usage)

- You can specify **shortopts** as a colon(:) separated single letter characters.
- You can specify **longopts** as a comma-separated list of words with the suffix **=**
- **longopts** without the suffix = are considered as **a flag** and they should be passed without any value.
- Now, to use the options passed to our program we can just iterate over the opts variable like any other list.

getopt (usage)

- Now, to use the options passed to our program we can just iterate over the opts variable like any other list:

```
for (o, a) in opts:
```

- Here **o** will hold our option name and **a** will have any value assigned to the option.

- Also notice that as **--subtract** is being used as a flag it will not have any value. This flag is used to decide whether to print the sum of the inputs **or** the difference in them.

getopt (usage)

```
artur@artur-MSI:~/Desktop/DCI$ python3 get_opt.py -a 6 -b 7

 Sum of two numbers is : 13

artur@artur-MSI:~/Desktop/DCI$ python3 get_opt.py -a 6 -b 7 --subtract

 Difference in two numbers is : -1
```

- Full example is available under the link in documentation!

argparse module

- The argparse module makes it easy to write user-friendly command-line interfaces.
- The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv.
- The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

argparse (example)

- The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

argparse (example)

Digital Career Institute

- Assuming the Python code above is saved into a file called prog.py, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

argparse (example)

- When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

argparse (example)

Digital Career Institute

- If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

- You can find more information about argparse module in the official tutorial (link in the **documentation**)

input() function

- Programs usually request for some user **input** to serve its function (e.g. calculators asking for what numbers to use, to add/subtract etc.).

- In Python, we request user input using the **input()** function.

```
>>> number = input("Type your first number: ")
Type your first number: 123
>>> print(number)
123
>>> print(type(number))
<class 'str'>
```

input() function

- This shown code is requesting for user input, and will store it in the **number** variable.
- **Note**: Inputs are automatically saved as **strings**.
- Therefore, always convert (cast) to proper type before doing any math operators like addition / subtraction.

At the core of the lesson

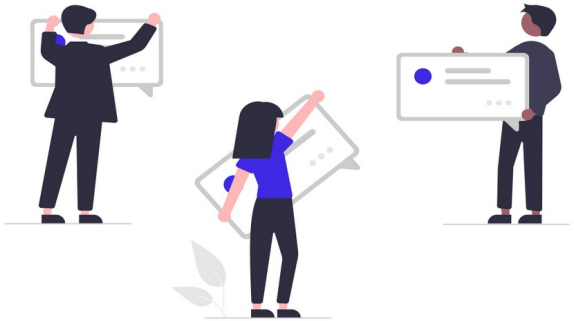
Digital Career Institute

Lesson learned:

- We know the idea of command line interface (CLI) and command line arguments
- We know usage of the sys.argv variable
- We know usage of the getopt module
- We know usage of the argparse module
- We know usage of the input() function

Self Study

Digital Career Institute



Topics

Expert Round

Digital Career Institute

At the core of the lesson

Digital Career Institute

Documentation

Documentation

1. Python tutorial on W3 Schools.
2. Guide to print() function in Python.
3. Python math module.
4. Python command line arguments.
5. The argparse basics.
6. Bitwise operators
7. Usage of getopt module

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute