Digital Career Institute

Python Course - Introduction

Digital Career Institute

Topics

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

Digital Career Institute

Hello world in Python with print()

- 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

```
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

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

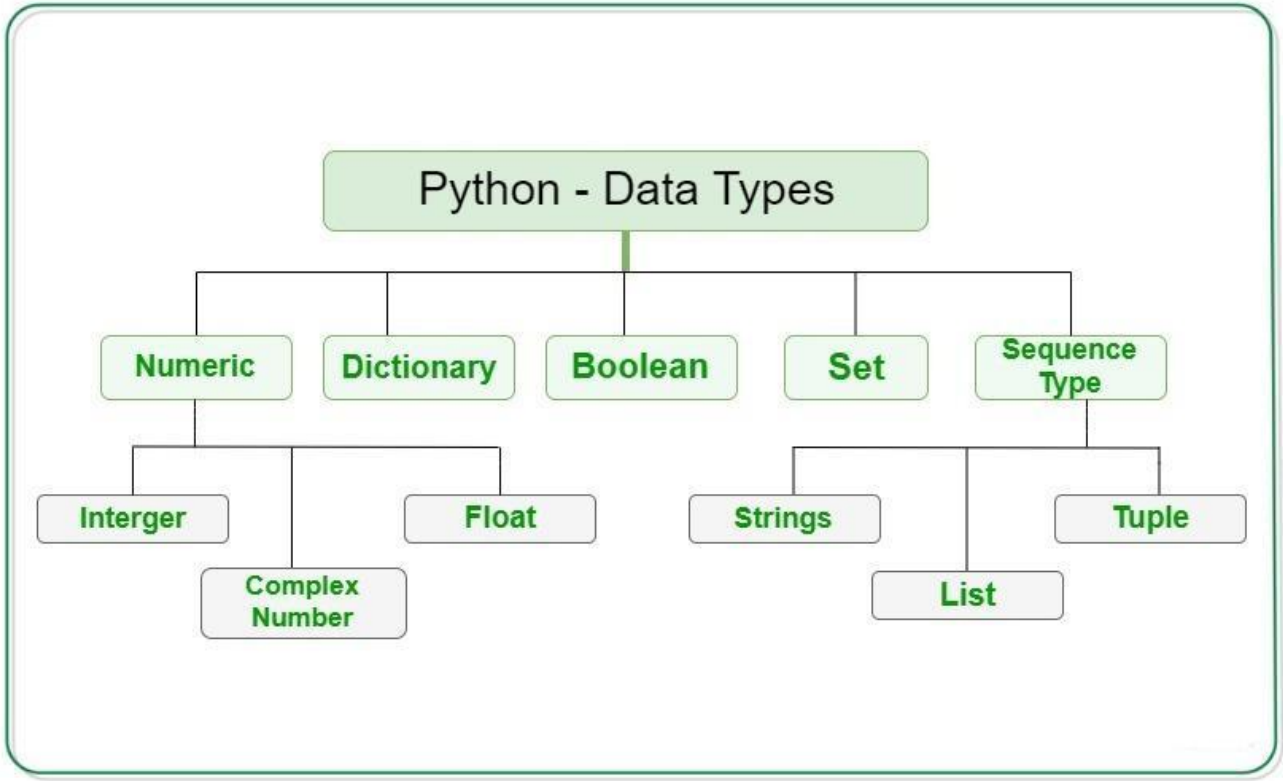
- You can assign the ***same*** value to **multiple variables** in one line:
  - x = y = z = 345
  - Now:
    - x = 345
    - y = 345
    - z = 345

Swapping variables

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

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

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()

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

```
>>> 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(12312312312312312312312312312312312312312312323233242345234342342342342342342342343437473847573847582374837847238482374872384783787473344345453434346903943992924384828489292844467364765635763756358556748564545487587584758745874857847574857847584758745784578457845748574857 + 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.

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

- 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

```
>>> 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

Digital Career Institute

Lesson learned:

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