Strathmore UNIVERSITY

# MASTER OF SCIENCE IN DATA SCIENCE AND ANALYTICS

## Topic 1: Intro to Programming

**DSA 8101: Fundamental Computing Concepts**

**Class Notes**

**Introduction to Python**

**About**: In this section we examine the essential building blocks of Python programming. We're going to start right from the beginning, so don't worry if you've never programmed before at all, this is the topic for you. We are going to work our way up from beginning programming concepts, all the way to being able to identify and handle errors. In this section we're going to focus on teaching you how to use Python as a simple calculator, while making effective use of the tools that we have available to us on Python. In other words, we want to learn not just how to program but also learn how to effectively use Python's extensive functionalities. In this course and throughout the class we are going to be using Python 3.10

# Using Python as a calculator

## Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages. For example:

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

```
>>> 8 / 5 # division always returns a floating point number
1.6
```

The integer numbers (e.g. 2, 4, 20) have type *int* , the ones with a fractional part (e.g. 5.0, 1.6) have type *float* . We will see more about numeric types later. Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # result * divisor + remainder
17
With Python, it is possible to use the ** operator to calculate powers:
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

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

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

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

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

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
```

```
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round (_, 2 )
113.06
```

In addition to *int* and *float* , Python supports other types of numbers, such as *Decimal* and *Fraction*. Python also has built-in support for complex numbers, and uses the *j* or *J* suffix to indicate the imaginary part (e.g. 3+5j).

## Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result 2 . \ can be used to escape quotes:

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

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn \' t," she said.'
'"Isn\'t," she said.'
>>> print ( '"Isn \' t," she said.' )
"Isn't," she said.
>>> s = 'First line. \n Second line.' # \n means newline
```

```
>>> s # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print (s) # with print(), \n produces a new line
First line.
Second line.
```

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

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

String literals can span multiple lines. One way is using triple-quotes: """..."""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. The following example produces the following output (note that the initial newline is not included):

```
print ( """ \
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
""" )
```

```
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

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

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

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
...
SyntaxError: invalid syntax
>>> ( 'un' * 3 ) 'ium'
...
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use +:

```
>>> prefix + 'thon'
'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[ 0 ] # character in position 0
'P'
>>> word[ 5 ] # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

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

Note that since -0 is the same as 0, negative indices start from -1. In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

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

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

```
>>> word[: 2 ] + word[ 2 :]
'Python'
>>> word[: 4 ] + word[ 4 :]
'Python'
```

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

```
>>> word[: 2 ] # character from the beginning to position 2 (excluded)
'Py'
>>> word[ 4 :] # characters from position 4 (included) to the end
'on'
>>> word[ - 2 :] # characters from the second-last (included) to the end
'on'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n* , for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0 1 2 3 4 5 6
- 6 - 5 - 4 - 3 - 2 - 1
```

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j* , respectively. For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2. Attempting to use an index that is too large will result in an error:

```
>>> word[ 42 ] # the word only has 6 characters
Traceback (most recent call last):
File "<stdin>" , line 1 , in <module>
IndexError : string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[ 4 : 42 ]
'on'
>>> word[ 42 :]
''
```

Python strings cannot be changed — they are *immutable* . Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[ 0 ] = 'J'
...
TypeError: 'str' object does not support item assignment
>>> word[ 2 :] = 'py'
...
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[ 1 :]
'Jython'
>>> word[: 2 ] + 'py'
'Pypy'
```

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

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

See also:

- **textseq** Strings are examples of *sequence types* , and support the common operations supported by such types.
- **string-methods** Strings support a large number of methods for basic transformations and searching.
- **f-strings** String literals that have embedded expressions.
- **formatstrings** Information about string formatting with str.format().
- **old-string-formatting** The old formatting operations invoked when strings are the left operand of the % operator are described in more detail here.

# Errors and Exceptions

## Syntax Errors

Syntax errors are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.
In IDLE, it will highlight where the syntax error is. Most syntax errors are typos, incorrect indentation, or incorrect arguments. If you get this error, try looking at your code for any of these.

## Logical Errors

These are the most difficult type of error to find, because they will give unpredictable results and may crash your program. A lot of different things can happen if you have a logic error. However these are very easy to fix as you can use a debugger, which will run through the program and fix any problems.

## Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * ( 1 / 0 )
Traceback (most recent call last):
File "<stdin>" , line 1 , in <module>
ZeroDivisionError : division by zero
>>> 4 + spam * 3
Traceback (most recent call last):
File "<stdin>" , line 1 , in <module>
NameError : name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>" , line 1 , in <module>
TypeError : Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are *ZeroDivisionError*, *NameError* and *TypeError* . The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it. The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

## Dealing with exceptions

Unlike syntax errors, exceptions are not always fatal. Exceptions can be handled with the use of a *try* statement.
The try statement works as follows.

- First, the *try* clause (the statement(s) between the *try* and *except* keywords) is executed.
- If no exception occurs, the *except* clause is skipped and execution of the *try* statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the *except* keyword, the except clause is executed, and then execution continues after the *try* statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

```
import urllib.request #for python 3
url = 'http://www.example.com'
try :
req = urllib2.Request(url)
response = urllib2.urlopen(req)
the_page = response.read()
print the_page
except :
print ( "We have a problem.")
```

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except ( RuntimeError , TypeError , NameError ):
... pass
try:
age = int ( input ("Enter your age: " ))
print ( "You must be {0} years old." .format(age))
except ValueError :
print ( "Your age must be numeric.")
```

If the user enters a numeric value as his/her age, the output should look like this:

```
Enter your age: 5
Your must be 5 years old.
```

However, if the user enters a non-numeric value as his/her age, a *ValueError* is thrown when trying to execute the int() method on a non-numeric string, and the code under the except clause is executed:

```
Enter your age: five
Your age must be numeric.
```