MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT9133 Semester 1 2019
Programming Foundations in Python

Week 2:
Python Basic Elements

Gavin Kroeger

GROUP OF EIGHT AUSTRALIA

Help for This Unit

- Lecture:
  - Ask me stuff
- Consultation
  - Mon 09:00-10:00, Caulfield-6-H695, Xinyu Li
  - Mon 16:00-17:00, Caulfield-6-H695, Gavin Kroeger
  - Wed 14:00-15:00, Caulfield-6-H695, Yiwei Zhong
  - Fri 10:00-11:00, Caulfield-6-H695, Shirin Ghaffarian Maghool
  - Fri 17:00-18:00, Caulfield-6-H695, Afsaneh Koohestani TO BE CONFIRMED
- PASS program
  - Unregistered students can also join
- Forum
  - Strongly recommended
    - Instant answers from tutors or other experienced students
    - Others can benefit from your questions.

Module 1 Synopsis

- This module is aimed to provide you with:
  - Overview of the Python programming language
  - Interactive IDE for Python
    - Jupyter notebook (a.k.a. iPython notebook)
  - Basic concepts of programming
    - Programs and algorithms, notion of abstraction
  - Fundamental programming constructs in Python
    - Primitive data types, variables, expressions, statements, assignments, arithmetic and logical operators, control structure

Module 1 Learning Outcomes

- After working your way through this module, you should be able to:
  - Recognize a computational problem
  - Define an algorithm for solving the problem
  - Identify and use various programming constructs used in Python
  - Efficiently write simple Python program with the correct grammar

Basic Elements of Python: Core Data Types

Core Data Types in Python

- Two types of (built-in) primitive data types supported by Python:
  - Atomic
  - Collective

- Atomic types:
  - *Indivisible* which represents only a single data value
  - E.g. Integer (type `int`), floating-point (type `float`), Boolean type (type `bool`)

- Collective types:
  - Comprised of more than one data values
  - E.g. String (type `str`)

Numbers in Python

- Two fundamental *numeric* representations:
  - Integers (type `int`)
  - Floating-point or real numbers (type `float`)

- Examples of valid numeric literals:

```
pos_number = 68
neg_number = -56
pi = 3.1416
gamma = 0.577215
another_number = 3.126e3
```

- Basic arithmetic operations supported:
  - Addition, subtraction, multiplication, and division

Booleans in Python

- Boolean (type `bool`):
  - Representation of logical values
  - Only two possible values: `True` or `False`
- Boolean type objects are *results* for comparison
  - Checking for equality of and relation between data values
  - E.g. less than `12 < 10`; greater than `12 > 10;` equality `12 == 21`;
- Standard logical operations supported:
  - `and, or , not`

It is ==, not =

Strings in Python

- String (type `str`):
  - A collection (sequence) of characters or letters
  - Any number of characters with a pair of quotation marks
    - ‘ ’ and “ ”

- Characters are any keystrokes:
  - Letter (‘a’), number (‘123’), punctuation (‘?’), space (‘ ’), tab (‘\t’), newline (‘\n’)

- Examples of valid string literals:

```
first_name = "Mary"
short_name = "M"
student_id = "12345678"
empty_str = " "
message = "Welcome to FIT9133!"
```

String Manipulation

- Common usage:
  - Presentation of computational results as some form of output
  - Using the built-in function `print()`

```
a = 1
b = 2
print("result is", a + b)
```

- Built-in methods for string manipulation:
  - `len()`
  - `String.upper()`
  - `String.lower()`
  - `String.count()`

```
>>> a_str = "hello"
>>> len(a_str)
>>>
>>> a_str.upper()
>>>
>>> a_str.lower()
>>>
>>> a_str.count('l')
>>>
```

https://docs.python.org/3/library/stdtypes.html#string-methods

String Manipulation

- Common usage:
  - Presentation of computational results as some form of output
  - Using the built-in function `print()`

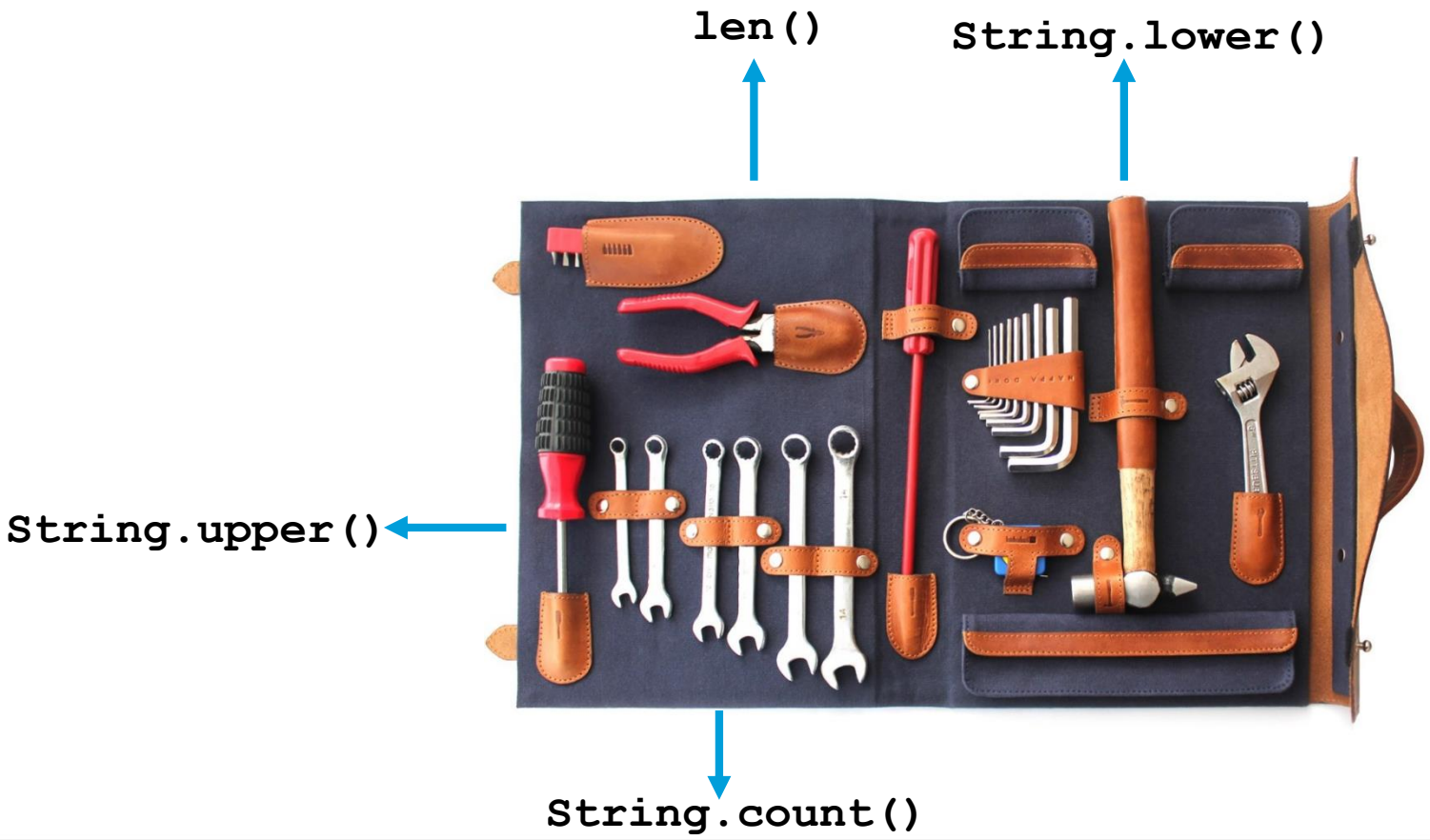
```
a = 1
b = 2
print("result is", a + b)
```

- Built-in methods for string manipulation:
  - `len()`
  - `String.upper()`
  - `String.lower()`
  - `String.count()`

```
>>> a_str = "hello"
>>> len(a_str)
>>> 5
>>> a_str.upper()
>>> 'HELLO'
>>> a_str.lower()
>>> 'hello'
>>> a_str.count('l')
>>> 2
```

https://docs.python.org/3/library/stdtypes.html#string-methods

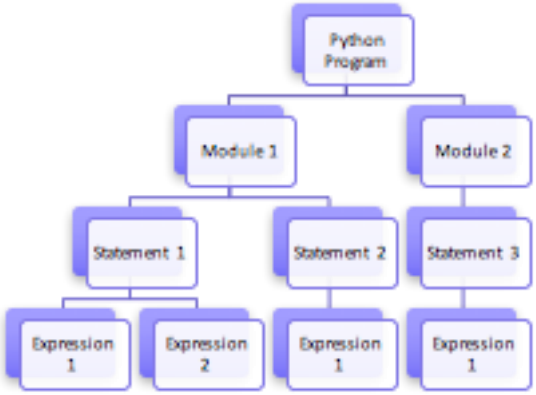
- Built-in methods for string manipulation



Basic Elements of Python: Operators and Expressions

The Composition of a Python Program

- A Python program contains one or more modules (i.e. Python source files)
- Each module contains one or more statements
- Each statement contains one or more expressions
- Each expression is composed of Python objects and operators



Operators and Expressions

- Expressions:
  - Composition of operators and data objects (i.e. data values)
  - Evaluated to a value of a specific data type
  - Evaluated results can be assigned to variables for further manipulation

- Basic syntax of an expression:

```
<operand> <operator> <operand>
```

- Operator:
  - Arithmetic operator
  - Relational operator
  - Logical or boolean operators

Arithmetic Operators

- Arithmetic operators:
  - Addition (`a + b`)
  - Subtraction (`a - b`)
  - Multiplication (`a * b`)
  - *Floor* division for integers (`a // b`)
  - *Real* division for floats (`a / b`)
  - Modulo/remainder (`a % b`)
  - Power (`a ** b`)

```
>>> 5 + 3
>>>
>>> 5 - 3
>>>
>>> 5 * 3
>>>
>>> 5 / 3
>>>
>>> 5 // 3
>>>
>>> 5.0 // 3
>>>
>>> 5 % 3
>>>
>>> 5 ** 3
>>>
>>> answer = 5 ** 3
>>> print(answer)
>>>
```

Arithmetic Operators

- Arithmetic operators:
  - Addition (`a + b`)
  - Subtraction (`a - b`)
  - Multiplication (`a * b`)
  - *Floor* division for integers (`a // b`)
  - *Real* division for floats (`a / b`)
  - Modulo/remainder (`a % b`)
  - Power (`a ** b`)

```
>>> 5 + 3
>>> 8
>>> 5 - 3
>>> 2
>>> 5 * 3
>>> 15
>>> 5 / 3
>>> 1.6666666666666667
>>> 5 // 3
>>> 1
>>> 5.0 // 3
>>> 1.0
>>> 5 % 3
>>> 2
>>> 5 ** 3
>>> 125
>>> answer = 5 ** 3
>>> print(answer)
>>> 125
```

Arithmetic Expressions

- Arithmetic expressions:
  - Composed by arithmetic operators and numbers (of type `int` or `float`)
  - Evaluated to a value of either `int` or `float`
  - If both of the operands are `int` , the result is usually an `int`
  - If one of the two operands is a `float`, the result is a `float`
- Order of operations (operator precedence):
  - Operators with higher precedence get calculated first
  - Operators with same precedence, calculation from left to right
  - `*` and `/` are higher precedence than `+` and `-`
  - Parentheses or brackets *override* any precedence

Question 4

What are the results of these following expressions that involve multiple arithmetic operators?

```
result = 12 - 5 // 11 + 2
result = (12 - 5) // 11 + 2
result = 12 - (5 // 11) + 2
```

A. 2, 14, 2

B. 14, 14, 2

C. 14, 2, 14

D. 2, 2, 14

Relational Operators

- Relational operators:
  - Equal (`a == b`)
  - Not equal (`a != b`)
  - Less than (`a < b`)
  - Less than or equal (`a <= b`)
  - Greater than (`a > b`)
  - Greater than or equal (`a >= b`)

Relational operators are used for determining the relationship that exists between two data values.

Relational Expressions

- Relational expressions:
  - Logical expressions evaluated to either **`True`** or **`False`**
  - Formed by relational operators and any data objects

- Evaluation:
  - If the object types are **`int`** or **`float`**, the values are compared based on the relative numerical order
  - If the object types are **`str`**, the values are compared based on the lexicographical order

```
>>> 23 < 123
>>>
>>> 'xyz' < 'xy'
>>>
>>> 123 == "123"
>>>
>>> 789 < "789"
>>>
```

Relational Expressions

- Relational expressions:
  - Logical expressions evaluated to either **`True`** or **`False`**
  - Formed by relational operators and any data objects

- Evaluation:
  - If the object types are **`int`** or **`float`**, the values are compared based on the relative numerical order
  - If the object types are **`str`**, the values are compared based on the lexicographical order

```
>>> 23 < 123
>>> True
>>> 'xyz' < 'xy'
>>> False
>>> 123 == "123"
>>> False
>>> 789 < "789"
>>> TypeError
```

(Compound) Relational Expressions

- Compound (or complex) relational expressions:
  - Operands are not restricted to a single value
  - Operands can be formed by arithmetic expressions

```
>>> 2 + 3 <= 7 - 2
>>>
>>> 5 / 2 == 5 // 2
>>>
>>> 6 / 2 == 6 // 2
>>>
```

(Compound) Relational Expressions

- Compound (or complex) relational expressions:
  - Operands are not restricted to a single value
  - Operands can be formed by arithmetic expressions

```
>>> 2 + 3 <= 7 - 2
>>> True
>>> 5 / 2 == 5 // 2
>>> False
>>> 6 / 2 == 6 // 2
>>> True
```

Arithmetic operators are of higher precedence than relational operators; hence arithmetic expressions are first evaluated and the comparison is then made on the resulting values.

Logical or Boolean Operators

- AND operator: `a and b`
  - Evaluated to `True` if and only if both `a` and `b` are `True`
- OR operator: `a or b`
  - Evaluated to `True` if either `a` or `b` is `True` or both `a` and `b` are `True`
- NOT operator:
  - To invert the Boolean value of the operand
  - If `a` is `True`: `not a` will turn `a` into `False`

Logical Expressions

- Logical operators are combined with relational operators to form compound logical expressions that are more complex

- Order of operations for compound expressions:
  - Logical operators are of the lowest order of precedence
  - Arithmetic expressions are first evaluated
  - Relational expressions are then evaluated before logical operators are applied

```
arithmetic operators >
                      relational operators >
                                            logical operators
```

(Compound) Logical Expressions

```
>>> x = 6
>>> y = 9
>>> x % 3 == 0 and x < 0
>>>
>>> x < 10 and x < y
>>>
>>> x + y > 10 or x + y < 10
>>>
```

(Compound) Logical Expressions

```
>>> x = 6
>>> y = 9
>>> x % 3 == 0 and x < 0
>>> False
>>> x < 10 and x < y
>>> True
>>> x + y > 10 or x + y < 10
>>> True
```

Logical expressions are often used as the conditions that control the *flow of execution* of a program, which are specified by the control structures defined within the program.

Basic Elements of Python: Statements and Assignments

Python Statements

- Statements:
  - Instructions (commands) of a Python program that are interpretable and executable by the Python interpreter

- Assignment statements:
  - Binding a data object (representing specific type of value) to a variable
  - Assigning the result of an expression to a variable

```
message = "Welcome to FIT9133"
temp_F = temp_C * 9 / 5 + 32
bool_result = value > 0 and value < 100
```

Single-Line Statements

- Single-line statements:
  - Each Python statement spans a single line of code

- You could split a single line statement across multiple lines
  - Append a backslash (‘\’) at the end of each line

```
bool_result =  value > 0 \
               and value < 100 \
               and value % 5 == 0
```

https://www.python.org/dev/peps/pep-0008/#maximum-line-length

Statement Blocks

- Statement blocks:
  - Certain programming constructs that span across multiple lines of code
  - Control structures that determine the flow of program execution

An Example of Statement Blocks

```
flag = True

if flag == True:
    print("YES")
    print("It is true!")

else:
    print("NO")
    print("It is false!")
```

- Either one of the statement blocks is executed based on a governing condition
- Important: the symbol ':' denotes the beginning of a statement block, i.e. the subsequent *indented* statements are part of that block
- Statement blocks are usually concluded with a new blank line

Indentation is semantically meaningful in Python programs.

Control Structures

Selection Constructs

- **`if-else`** statements:
  - Using logical expressions as selection conditions to determine which statement block to be executed

```
if the condition is True:
    do this statement block
else:
    do this statement block
```

- Note: Indentation is important in defining the "scope" of a block of statements.

```
message = "Welcome to FIT9133"
letter = 'o'
count = message.count(letter)
if count < 1:
    print(letter + " doesn't exist in " + message)
else:
    print(letter + " exists in " + message)
    print(letter + " occurs " + str(count) + " times")
```

- Nested **`if`** statements:
  - Useful for when multiple conditions need to be considered

```
message = "Welcome to FIT9133"
letter = 'o'
count = message.count(letter)
if count < 1:
    print(letter + " doesn't exist in " + message)
else:
    print(letter + " exists in " + message)
    if count >= 5:
        print(letter + " occurs 5 times or more")
    else:
        print(letter + " occurs less than 5 times")
```

Selection Constructs

- **`elif`** statements:
  - Similar to nested **`if`** statements; combining an **`if`** with an **`else`**

```
message = "Welcome to FIT9133"
letter = 'o'
count = message.count(letter)
if count < 1:
    print(letter + " doesn't exist in " + message)
elif count >= 5:
    print(letter + " exists in " + message)
    print(letter + " occurs 5 times or more")
else:
    print(letter + " exists in " + message)
    print(letter + " occurs less than 5 times")
```

- **`while`** loop:
  - A block of statements will be executed repeatedly as long as the governing condition is **`True`**

```
while the condition is True:
    do this statement block
```

- Note: The governing condition (logical expression) has to turn into **`False`** eventually; otherwise the loop will run *infinitely*.

```
number = 0
while number < 5:
    number += 1
    print(number)
```

- **`for`** loop:
  - Similar to **`while`** loop; except that the governing condition (logical expression) does not need to be defined
  - Useful for iterating or traversing through a collection of items (e.g. Lists)

```
num_list = [1, 2, 3, 4, 5]
product = 1
for item in num_list:
    product *= item
print(product)
```

- Note: This special **`for`** loop structure with the **`in`** operator can be used on any *iterable* collective data type (e.g. Dictionary).

MONASH
University

Standard Input and Output in Python

Input and Output

- Input and output:
  - Two essential components of a program
- Input:
  - Data needed for solving a specific computational problem
- Output:
  - Presentation of the computational results

```
a = 1
b = 2
result = a + b
print("The addition of a and b is", result)
```

Standard Input

```
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
result = a + b
print("The addition of a and b is", result)
```

- To obtain data values through the standard input (i.e. the keyboard)
- The **`input`** function:
  - **`input("prompt statement:")`** or **`input()`**
  - Python built-in function to obtain data externally
  - Input values are returned as objects of type **`str`**
  - Convert input values into type **`int`** using the conversion function **`int()`** in order to perform arithmetic operations

type casting or type conversion

Standard Output

- To display any information and or computational results on the standard output (i.e. the terminal screen or console)

- The **`print`** function:
  - **`print("output string")`**
  - By default a newline character ('**`\n`**') is appended at the end
  - Each print statement will display the output on a separate line
  - Output arguments to be displayed must of of type **`str`**

```
print("This is just a print statement.")
print("This is another print statement.")
```

What would be the output on the screen?

Question 4

What would be the output on the screen?

```
print("This is just a print statement.", end = " ")
print("This is another print statement.")
```

A. Two separate lines
B. One single line
C. Not sure

Standard Output

- Two ways of displaying multiple output arguments with **`print()`**:
  - With the comma '**,**' to separate each output argument
  - With the operator '**+**' to concatenate multiple output arguments

```
print("The addition of a and b is", result)
print("The addition of a and b is " + str(result))
```

type conversion

- Note: When '**+**' is used with output arguments of Python built-in types (**`int`** or **`float`**), explicit type conversion to **`str`** is required.

Opening Files

- Opening a file: **`open(file,mode)`**
  - Create a file handle as a reference to the file to be handled
  - Two major String-typed arguments:
    - **`file`**: the name (or the path name) of a file
    - **`mode`**: the kind of operation mode that the file should be opened in

- Open modes (from Python 3 documentation):

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)

File Input

- Reading lines from a file: `input_handle = open(file, 'r')`
  - `input_handle.readline()`: read one line at a time ('`\n`' is included) until the end of file is reached
  - `for line in input_handle`: iterate through each of the lines on each iteration of the loop
  - `input_handle.readlines()`: read the entire content of a file and return as a list
  - `input_handle.read()`: read the entire content of a file and return as a string

- Closing a file: `input_handle.close()`
  - "Good practice" is to close all the files when they are no longer needed to be accessed

Files are usually closed when the Python program terminates.

File Output

- Open modes for file writing:
  - `open(file_handle, 'w')`: overwrite the existing content of the output file with the new content
  - `open(file_handle, 'a')`: append the new content at the end of the output file

- Writing lines to a file:
  - `file_handle.write(the_line)`: write one line at a time to the file
  - Note: '`\n`' is often appended at the end of each line before writing to the file

Each line to be written to the output file is a single string.