Chapter 1 - Introduction to Python

CS 171 - Computer Programming 1
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Outline

Getting Started

Python code

- can be written in any **plain** text editor
  - Vim/Emacs are plain text editors (many others exist depending on the Operative System you use)
  - MS Word/Wordpad/LibreOffice Writer are **not** plain text editors
- usually has the `.py` extension

Let's create a file, e.g., named `hello.py`, with content:

```
# Our first Python program

print ("Hello", "World!")
```

Getting Started

Let's create a file, e.g., named `hello.py`, with content:

```
# Our first Python program

print ("Hello", "World!")
```

We will assume that all the code is saved in:

- On Windows

  `C:\py3eg`

- On Unix/Linux/Mac OS

  `$HOME/py3eg`

Getting Started

Running Python code

- Python programs are executed by the Python interpreter
  - Usually inside a console window
    - On Windows, the console is called *Console* or *DOS prompt*, or similar
    - On Mac OS X the console is provided by the *Terminal.app*

Start up a console, and:

- Change the current directory to the directory where you saved `hello.py` and run it:
  - On Windows

    ```
    C:\>cd c:\py3eg
    C:py3eg\>python.exe hello.py
    ```

  - On Mac OS

    ```
    $ cd $HOME/py3eg
    $ python3 hello.py
    ```

Getting Started

Let's check the output:

```
$ Hello World!
```

Why was that the output?

- Lines starting with # are comments
- The second line is blank, which is ignored
- The third line is actual Python code
  - `print` is called with two string arguments
- Each statement is executed in turn, from the first to the last
- The program terminates

Getting Started

About `print`

- it is a built-in part of Python
  - we used it without needing to define it
  - and without importing or including it from a library
- it can take many arguments
- it separates each argument with one white space
- it prints a newline at the end

Outline

Getting Started

- Python is built around 8 central pieces
- We will start by introducing each such piece
  - using simple but realistic programs
- later we will dedicate more time to each of them

1. Data Types

Data Types

- Are fundamental to all programming languages;
- Are needed to represent items of data;
- Python provides several built-in data types
  - Focusing on *just* two for now:
    - integers
    - strings

1. Data Types

Integers

- Are represented using the `int` type;
- Examples include:

```
-973
12343263524152363472453
0
```

1. Data Types

Strings

- Are represented using the `string` type;
- Examples include:

```
"This is a sequence of characters"
'John Doe'
'123'
'and me as well !"#%'
''
```

- can be delimited by single or double quotes, but use the same at both ends;
- can be empty (last example);

1. Data Types

About sequences

- Python uses square brackets `[]` to access an item from a sequence such as (but not limited to) a string;
- Positions start at 0;
- From now on, every time you see >>>, the text that follows it is being executed inside a Python shell;

```
>>> "Hard Times"[5]
'T'
>>> "giraffe"[0]
'g'
```

1. Data Types

In Python

- `str` and basic numeric types such as `int` are immutable;
  - once set, their value cannot be changed;
- This means, for example, that:
  - you can use `[]` to retrieve a character from a string but not to set a character in a string;

Data types can be converted:

- using *datatype(item)*:

```
>>> int("45")
45
>>> int("   45 ")
45
>>> str(912)
'912'
```

2. Object References

- Once we have data types, we need variables in which to store them;
- Python does not have variables as such, but *Object References*;

```
x = "blue"
y = "green"
z = x
```

- To execute the first example, Python creates a `str` object with the text *blue* and creates an object reference called `x` to the `str` object;
- Second example is similar;
- The third example creates a new object reference called `z` and sets it to refer to the same object that the `x` object reference refers to (in this case the `str` containing the text *blue*);

2. Object References

The = operator

- it is not the same as the variable assignment operator in some other languages;
- = binds an object reference to an object in memory;

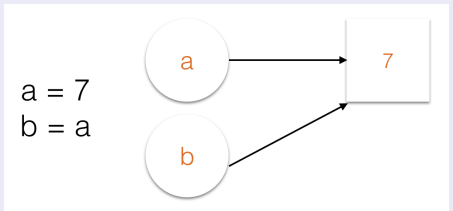
```
x = "blue"
y = "green"

z = x
print (x, y, z) # prints: blue green blue

z = y
print (x, y, z) # prints: blue green green

x = z
print (x, y, z) # prints: green green green
```

2. Object References

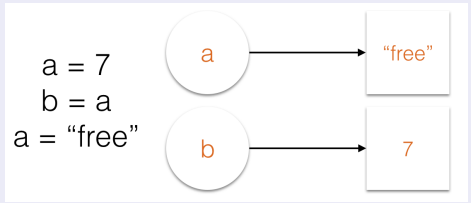
It is very important that you understand Object References

- Let's create an object reference to an object:

- If you add another object reference to the object a is referencing:



- If you now edit the object reference a to reference a new object:



2. Object References

Names for Object References, also called Identifiers:

- may not be the same as any Python keyword;
- must start with a letter or an underscore and be followed by zero or more nonwhite-space letter, underscore, or digit;
  - there is no length limit
  - UNICODE letters and digits
- are case-sensitive
  - `LIMIT` and `Limit` and `limit` are three different identifiers

2. Object References

Python uses dynamic typing

- an object reference can be rebound to refer to a different object;
- including of a different type;

```
route = 866
print(route, type(route)) # prints: 866 <class 'int'>
route = "North"
print(route, type(route)) # prints: North <class 'str'>
```

- when we reuse the `route` object reference to refer to a new `str`, the `int` object is scheduled for garbage collection since no object reference refers to it;

3. Collection Data Types

- Often we need to manipulate collections of data items;
- Python provides many collection types, but for now we focus on `tuple` and `list`;
- Both can be used to hold any number of data items of any data type;

3. Collection Data Types

The `tuple` data type

- Tuples are immutable;
  - Once they are created we cannot change them;
- Are created using commas `(,)`

```
>>> "Denmark", "Finland", "Norway", "Sweden"
('Denmark', 'Finland', 'Norway', 'Sweden')

>>> ("one",)
('one',)

>>> "one",
('one',)
```

- Python always outputs a tuple enclosed in parentheses;

3. Collection Data Types

The `list` data type

- List are mutable;
- Lists can be created using square brackets `[]`

```
[1, 3, 5, 7, 9]

['alpha', 'bravo', 'charlie', 'delta', 'echo']

['zebra', 49, -879, 'dark']

[]
```

- The last example is the empty list

3. Collection Data Types

Lists and Tuples

- Don't store data items at all, but rather object references;
- When they are created, they take copies of the object references they are given;
- Are objects, so we can nest them inside other collection data types:

```
[(1, '1'), ["abc"]]

(((1, 2), 3), [1, 2, 3, 4])
```

3. Collection Data Types

- One important Python function is `len()`,
  - takes a single data item as argument
  - returns the length of the item, i.e., the number of items in the data, as an `int`;

```
>>> len (("one", ))
1

>>> len ([3, 5, 1, 2, "pause", 5])
6

>>> len ([3, [5, 1, 2], "pause", 5])
4

>>> len ("count")
5
```

3. Collection Data Types

- All Python data items are objects (also called *instances*) of a particular type (also called a *class*);
- We will use *data type* and *class* interchangeably;
- Objects can have associated methods; for example, the `list` type has an `append()` method:

```
>>> x = ["zebra", 49, -879]
>>> x.append("more")
>>> x
['zebra', 49, -879, 'more']
```

- One can also make the method type explicit:

```
>>> list.append(x, "and more")
>>> x
['zebra', 49, -879, 'more', 'and more']
```

3. Collection Data Types

- Other methods are available for lists, e.g.,
  - `insert()`, which inserts an item at a given index position;
  - `remove()`, which removes an item at a given index position;
  - Always remember that indexes start at 0.

Lists and Tuples are sequences, so we can directly access elements:

```
>>> x
['zebra', 49, -879, 'more', 'and more']
>>> x[0]
'zebra'
>>> x[4]
'more'
>>> y = (1, 2, "abc")
>>> y[0]
1
>>> y[2]
'abc'
```

3. Collection Data Types

Lists and Tuples are sequences, so we can directly access elements:

```
>>> x
['zebra', 49, -879, 'more', 'and more']
>>> x[0]

>>> y = (1, 2, "abc")
>>> y[2]
'abc'
```

But since Lists are mutable and Tuples are immutable, we can use square brackets to set list elements, but not tuple elements:

```
>>> x
['zebra', 49, -879, 'more', 'and more']
>>> x[0] = 1
>>> x
[1, 49, -879, 'more', 'and more']
```

4. Logical Operations

- Logical operations are fundamental to any programming language;
- Python provides 4 sets of logical operations;
  - The Identity Operator
  - Comparison Operators
  - The Membership Operator
  - Logical Operators

4. Logical Operations

The Identity Operator

- The `is` operator checks if two object references refer to the same object:

```
>>> a = ["Retention", 3, None]
>>> b = ["Retention", 3, None]
>>> a is b
False
>>> a = b
>>> a is b
True
```

- Note that `is` is not used to compare the *values* of objects.
- `is` is often used to compare a data item with the built-in null object, `None`:

```
>>> a = "Something"
>>> b = None
>>> a is not Note, b is None
(True, True)
```

4. Logical Operations

Comparison Operators

- Python provides the standard set of binary comparison operators:
  - < less than
  - <= less than or equal to
  - == equal to
  - != not equal to
  - >= greater than or equal to
  - > greater than
- These operators compare object values;

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a <= b, a != b, a >= b, a > b, 0 <= a <= 10
(True, True, False, False, True)
```

4. Logical Operations

The Membership Operator

- We can test element membership For data types that are sequences or collections
  - such as strings, list and tuples
- we use `in` to test membership;
- and `not it` to test nonmembership;

```
>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True
```

4. Logical Operations

The Membership Operator

- For lists and tuples, `in` uses linear search, which can be slow for large collections;
- `in` is very fast when used in a dictionary or a set (we will study these later);
- `in` can also be used on strings:

```
>>> phrase = "Wild Swans by Jung Chang"
>>> "J" in phrase
True
>>> "not" in phrase
False
```

4. Logical Operations

Logical Operators

- Python provides three logical operators:
  - `and`
  - `or`
  - `not`
- These operators can be applied not only to Booleans

4. Logical Operations

`and` and `or` use short circuit logic

- `x and y` is equivalent to *if x then y else x* (note: this is not Python code)
- `x or y` is equivalent to *if x then x else y*
- in a Boolean context,
  - `0` is interpreted as `False` and all the other `ints` as `True`
  - `""` is interpreted as `False` and all the other `strings` as `True`

```
>>> 0 and 5
0
>>> 5 and 2
2

>>> 1 or 0
1
>>> 0 or 5
5
```

5. Control flow Statements

- Instructions in a code file are executed sequentially, but
- the flow of control of a program can be diverted by a function or method call or by a control structure such as
  - a conditional branch
  - a loop statement
- For now, we look at
  - `if`
  - `while`
  - `for`

5. Control flow Statements

About Boolean Expressions in Python

- `True` and `False` are booleans;
- the special object `None`, empty sequences or collections, or a numeric data item of value `0` all evaluate to `False`
- anything else is considered to be `True`

5. Control flow Statements

The `if` Statement

```
if _boolean_expression1_:
    _suite1_
elif _boolean_expression2_:
    _suite2_
...
elif _boolean_expressionN_:
    _suiteN_
else:
    _else_suite_
```

- There can be zero or more `elif` clauses;
- The final `else` is optional;
- Indentation is mandatory (Python avoids the use of parenthesis or braces)

5. Control flow Statements

The if Statement

```
if _boolean_expression1_:
    _suite1_
elif _boolean_expression2_:
    _suite2_
...
elif _boolean_expressionN_:
    _suiteN_
else:
    _else_suite_
```

- ... are used to indicate lines that are not shown
- a *suite* is a sequence of one or more statements
- pass is a special statement that does nothing (to be used, e.g. when a statement is mandatory but we want to do nothing)

5. Control flow Statements

The if Statement

```
if x:
    printf("x is nonzero")

if lines < 1000:
    print("small")
elif lines < 10000:
    print("medium")
else:
    print("large")
```

5. Control flow Statements

The `while` Statement

- Is used to execute a suite zero or more times;
- A simplified version of the `while` loop's syntax is (back to this later):

```
while _boolean_expression_:
    _suite_
```

- A `while` loop with a very typical structure is:

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

- `break` switches control to the first statement outside the current loop;
- a `continue` instruction inside a loop ignores the instructions until the current iteration of the loop, and switches control to the start of the loop;

5. Control flow Statements

The `for ...in` Statement

- Python's `for` loop reuses the `in` keyword:

```
for _variable_ in _iterable_:
    _suite_
```

- An `_iterable_` is any data type that can be iterated over
  - includes strings, lists and tuples

```
countries = ["Denmark", "Finland", "Norway", "Sweden"]
for country in countries:
    print(country)
```

- Note that the above result is sort of equivalent to `print(countries)`, but the output is not really the same (please run both and check);

5. Control flow Statements

Basic Exception Handling

- Many of Python's functions/methods indicate errors by raising exceptions;
- An exception is an object like any other Python object,
- and when converted to a string, produces a message text.

```
try:
    _try_suite_
except _exception1_ as _variable1_:
    _exception_suite1_
...
except _exceptionN_ as _variableN_:
    _exception_suiteN_
```

5. Control flow Statements

Basic Exception Handling

- the `as _variable_` part is optional: we may only care that a particular exception was raised and not be interested in its message text;
- If all the statements in the `try` suite execute *properly* (i.e., without raising an exception), the `except` blocks are skipped;
- If an exception is raised, control is immediately passed to the suite corresponding to the first matching `_exception_`
  - and any statements in the suite that follow the one that caused the exception will not be executed.
- If an exception occurs, and if the `_variable_` part is given, then inside the exception-handling suite, `_variable_` refers to the exception object;

5. Control flow Statements

Basic Exception Handling

- An example is as follows:

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered: ", i)
except ValueError as err:
    print(err)
```

- If the user enters 3.5, the output will be:

```
invalid literal for int() with base 10: '3.5'
```

- But if a valid integer was entered, e.g., 13, the output would be:

```
valid integer entered: 13
```

6. Arithmetic Operators

- Python provides a full set of arithmetic operators, including for the 4 mathematical operations:
  - +, -, * and /

```
>>> 5+6
11
>>> 3-7
-4
>>> -3
-3
```

- the division operator produces a floating-point value, not an integer;

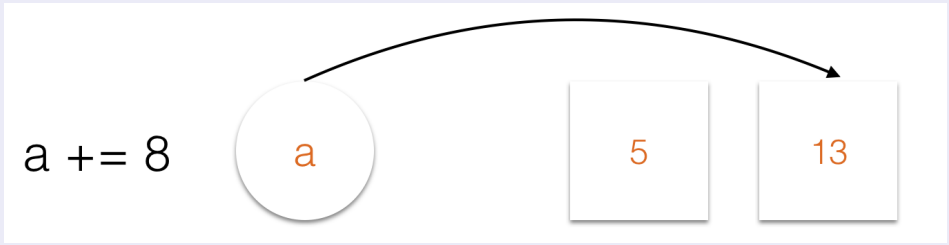
```
>>> 12/3
4.0
>>> 3/2
1.5
>>> int(12/3)
4
```

6. Arithmetic Operators

- There are also augmented assignment operators such as += and *=;

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

- But remember that `ints` are immutable, so



7. Input/Output

- Reading from input/writing to output is often key to write useful programs;
- We already mentioned and seen at work the `print()` function;
- Python also provides an `input()` function to accept input from the user;
  - it takes an optional string argument, that is printed on the console;
  - it waits for the user to type in a response and to finish pressing `Enter`;
  - if the user does not type any text but just presses `Enter`, the empty string is returned;
  - otherwise, a string containing what the user typed is returned, without any line terminator.

7. Input/Output

```
print("Type integers, each followed by Enter; or Enter to finish")

total = 0
count = 0

while True:
    line = input("integer: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break
if count:
    print("count =", count, "total =", total, "mean =", total/count)
```

7. Input/Output

- A typical run of this program looks like:

```
Type integers, each followed by Enter; or Enter to finish
number: 12
number: 7
number: 1x
invalid literal for int() with base 10: '1x'
number: 15
number: 5
number:
count = 4 total = 39 mean = 9.75
```

- Notice that the program was able to proceed upon erroneous situations.

8. Creating and Calling Functions

- Python allows us to define functionality to serve our specific purposes.
  - e.g., as a means to encapsulate suites as functions that can be parameterized by their arguments;
- The general syntax for creating a function is:

```
def _functionName_ (_arguments_):
    _suite_
```

- `_arguments_` are optional and multiple arguments must be separated by comma;
- Every function has a return value;
  - which is `None`, unless the function includes a `return _value_` statement;
  - the return value can be just one or a tuple of values;
  - the return value can be ignored by the caller;

8. Creating and Calling Functions

- Here is a function to obtain an integer from the user:

```
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

- This function can be used in a simple way:

```
age   = get_int("enter your age: ")
...
age_d = get_int("how old is your daughter? ")
```