Lecture 1: Introduction to Python

CMPUT 275

Winter 2018

Style Guidelines

PEP 8 is the de-facto code style guide for Python and you are expected to follow it in this course. You can use the pep8 commandline tool to check your Python code against some of the style conventions in PEP 8 and format it correctly.

```
$ pep8 --show-source --show-pep8 <program>.py
```

Where `<program>` should be substituted with the name of the Python program (aka "script") you created. The program autopep8 can be used to automatically reformat code in the PEP 8 style. Use it to format a file in-place with:

```
$ autopep8 --in-place <program>.py
```

Interpreted vs. compiled language

Interpreting a language provides some additional flexibility over compiled implementations. Features that are often easier to implement in interpreters than in compilers include:

- platform independence
- **dynamic typing**
- smaller executable program size (since implementations have flexibility to choose the instruction code)
- reflection (you can find out about the type, class, attributes and methods of an object)

What are the disadvantages of interpreted languages?

- Without static type-checking, which is usually performed by a compiler, programs can be less reliable, because type checking eliminates a class of programming errors
- Slower execution compared to direct native machine code execution on the host CPU
- Source code can be read and copied, or more easily reverse engineered in applications where intellectual property has a commercial advantage

Dynamic typing vs. static typing

- Statically-typed languages require you to declare the data types of your variables before you use them, while dynamically-typed languages do not
- Dynamically-typed languages perform type checking at runtime, while statically typed languages perform type checking at compile time

Structuring with Indentation

Programming languages usually use certain methods to group statements into blocks:

- begin ... end (e.g., Pascal)
- do ... done (e.g., bash shell)
- if ... fi (e.g., bash shell)
- indentation (e.g., Python)

In Python, all statements with the same distance to the left belong to the same block of code, i.e. they are vertically aligned. The block ends at a line less indented or the end of the file. If a block has to be more deeply nested, it is simply indented further to the right.

NOTE: Use TAB for indentation, do not even think of using spaces, not mentioning mixing spaces and tabs!

Python 2 vs. Python 3

Open a terminal and type:

```
$ python --version
$ python3 --version
```

A non-exhaustive list of features which are only available in 3.x releases and won't be backported to the 2.x series:

- print is a function; thus, we have to wrap the object that we want to print in parantheses
- integer division is supported in 3.x
- strings are utf-8 Unicode by default
- clean Unicode/bytes separation
- exception chaining
- function annotations (syntax for adding arbitrary metadata to Python functions)
- extended tuple unpacking
- non-local variable declarations

Python 3.x introduced some Python 2-incompatible keywords and features that can be imported via the in-built `__future__` module in Python 2. It is recommended to use `__future__` imports it if you are planning Python 3.x support for your code.

NOTE: Your IPython notebook uses Python 3 kernel!

The Basics of Python

Here is a quick-reference summary of the python tools and syntax learned in Lecture 1.

Running a python script

The Python interpreter (e.g., python3) compiles your program to the internal python byte-code which is then executed. To run a basic python program, type the following into the terminal, where `<program>` should be substituted with the name of the Python script you wish to run.

```
$ python3 <program>.py
```

To create a new file in your text editor, use:

```
$ atom -n <program>.py
```

or use this if you wish to try using Sublime, a text editor that is newly installed on the CMPUT 274/275 VM:

```
$ subl -n <program>.py
```

A "Hello World" program in Python

```python
# print() can be used to print a string to standard output
print("Hello World!")
```

Python Syntax

- Comments in python can be single or multiline.
  - Single-line comments are started with "`#`"
  - Multi-line comments are enclosed between pairs of three single apostrophes
- Variables and constants:
  - A new variable is commonly assigned using:
    `<identifier> = <expression_giving_a_value>`
- Rules for creating an identifier:
  1. Must start with a letter or underscore (`_`), but cannot start with a digit!
  2. Can be followed by any number of letters, digits, or underscores
  3. Cannot be a reserved word. Reserved words include

and	del	from	not	while	as
elif	global	or	with	assert	else
if	pass	yield	break	except	import
print	continue	class	exec	in	raise
finally	def	for	lambda	try	return

- Operators are pretty much the same as C/C++
  - Arithmetic operators: `+, -, *, /, %, **, //`
  - Bitwise operators: `&, |, ^, ~, <<, >>`
  - Comparison Operators: `==, !=, <, <=, >, >=`
  - Logical Operators: `and, or, not`

- Built-in types in Python include str, int, float, list, tuple, dict, and set. However, because python is dynamically typed, we do not declare these types when defining variables. To determine the type of an object, use the `type()` function
- Numeric types include integers, floating point numbers, and complex numbers

Lists

Lists in Python are like arrays in C, except that they can hold data of any type (actually, they hold addresses to objects) and their size can be dynamically changed. The syntax is:

`mylist = list()` or `mylist = []`

Negative indexing can be used to start indexing from the end of a list:

e.g. `mylist[-1]` returns the last element of mylist

Lists can be **concatenated using + or duplicated using ***:

- eg, `mylist + mylist2`
- `mylist * 5`

Unpacking: We can use the * operator to unpack the elements out of a list.

e.g. `first_element, *rest = listA` separates the first element from the rest of the list.

Other list operations:

- list.append(X) adds X to the end of the list
- list.insert(i, X) adds X at position i
- list.extend(L) adds a list L of items to the end
- list.remove(X) removes the first occurence of X
- list.pop(i) deletes and returns item list[i], while list.pop() deletes and returns the last item
- del list[i] deletes the ith item of list (Note this is a "del statement", not a method)
- list.reverse() reverses the list
- list.sort() sorts the list

Dictionary – Mapping Types

A dictionary is a mapping object, which maps values to arbitrary objects. Some syntax:

```python
# Define a dictionary. The second method only works when keys are strings.
some_dict = {'one': 1, 'two': 2, 'three': 3}
another_dict = dict(one=1, two=2, three=3)

# Assigning a key-value pair in the dictionary:
some_dict["one"] = 100

# To delete a key value pair:
del some_dict["one"]

# Return a list of all the keys used in a dictionary in a arbitrary order:
keys = list(some_dict.keys())

# Return a list of all the values, where again the order is arbitrary:
values = list(some_dict.values())
```

Functions

To **call a function** you always write its name, followed by some arguments in parentheses. The function does some action depending on its arguments. When there are multiple arguments to a function, you separate them with commas.

```python
# Defining a fuction
def do_nothing(some_input):
    # next line must be indented
    pass
```

The **pass statement** in Python is used when a statement is required syntactically but you do not want any command or code to execute.

```python
# Calling a function
some_arg = 10
do_nothing(some_arg)
```

An underscore can be used to **ignore** one or more output variables.

```python
def return_multiple_vars(first_input, second_input):
    return first_input + second_input, first_input - second_input

# Ignoring output variable
first_output, _ = return_multiple_vars(15, 10)
_, second_output = return_multiple_vars(15, 10)
```

The function `locals()` can be used to update and return a dictionary representing the local symbol table. It consists of `{local_variable_name : value}` pairs.

Required arguments vs Keyword arguments

```python
def compute_speed(distance, time):
    print("distance covered in km:", distance)
    print("time taken in h:", time)
    speed = distance / time
    return speed

# Required arguments
print_speed(compute_speed(101, 2))

# Keyword arguments
print_speed(compute_speed(time=2, distance=101))
```

Default arguments indicate that the function argument will take that value if no argument value is passed during function call.

```python
def compute_speed(distance, time=1): # time has a default value
    speed = distance / time
    return speed

speed = compute_speed(100) # time need not be specified
```

The operator * can be used to **specify a variable number of arguments**. This is useful when we do not know the exact number of arguments in advance that will be passed to the function.

```python
def say_hello(*varargs):
    for name in varargs:
        print("Hello", name)
```

Function annotations can be used for documentation or for pre-condition/post-condition checking. Annotations are not actually enforced.

```
def div(a: 'the dividend defaults to 1',
    b: 'the divisor (must not be 0)') -> 'the result of dividing a by b':
    """Divide a by b"""
    return a / b

for arg in div.__annotations__:
    print(div.__annotations__[arg])
```

Sometimes it is acceptable to break a long line of code into multiple shorter ones to avoid text wrapping at an inconvenient spot. You can use Python's **implicit continuation** (as in the example above) by placing all the divided lines within a single set of brackets, braces, or parentheses. Here's another example:

```
if (1000 < year < 2100 and 1 <= month <= 12
   and 1 <= day <= 31 and 0 <= hour < 24
   and 0 <= minute < 60 and 0 <= second < 60):   # implicit continuation
        return True
```

Python also allows for **explicit continuation**. The backslash, "\", at the end of each line indicates a line break that continues the expression. This improves readability when code is too long to fit on a single line, but is not always required.

```
if 1000 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # explicit continuation
        return True
```

See this section of the PEP8 Style Guidelines for more information and examples.

Additional Notes

- Python is case-sensitive and uses 0-based indexing
- Python does not require semi-colons to terminate statements. You just need to hit return and then Tab on the first line. Hence, Python both enforces and standardizes indentation
- **How does variable assignment work?**: Python first creates the value corresponding to the expression on the right-hand side of the assignment operator in the memory. This means allocating a little (or more) memory as needed, putting the type next to the value and taking the address of the start of the allocated chunk. Next it checks its namespaces, or frames (which essentially store name-address associations) to see whether the given name exist (in reality, sometimes but not always the names are replaced in a precompilation phase by numbers, but the idea is the same, so let's still think that python deals with the names we write in our codes). If the name exists, its associated address is replaced by the new address of the value created. If it does not exist, a new entry is created where the address is stored against next to the name

Python vs. C/C++

Main Differences

- **Memory management:** Unlike Python, C++ doesn't have garbage collection, and encourages use of raw pointers to manage and access memory. Hence, it requires much more attention to bookkeeping and storage details, and while it allows you very fine control, it's often just not necessary
- **Types:** C++ types are explicitly declared, bound to names, checked at compile time, and strict until they're not. Python's types are bound to values, checked at run time, and are not so easily subverted. Python's types are also an order of magnitude simpler. Python has high-level native data types (strings, tuples, lists, sets, dictionaries, file objects, etc.) The safety and the simplicity and the lack of declarations help a lot of people move faster
- **Language complexity:** Even the best C++ developers can be caught up short by unintended consequences in complex (or not so complex) code. Python is much simpler, which leads to faster development and less mental overhead
- **Syntax:** Python has clean, straightforward syntax compared to C++
- **Interpreted vs. compiled:** C++ is almost always explicitly compiled. Python is not (generally). It's common practice to develop in the interpreter in Python, which is great for rapid testing and exploration
- Python has great support for building web applications
- Python has a huge standard library with many built-in functions

Some Similarities

- Both support object oriented programing paradigms
- Both have exceptions
- Both have concurrency support