INFO1910 S2 2023 Week 8 Tutorial

Introduction to Python

Introduction to Python

Python is an interpreted scripting language. It's a nice glue for quickly knocking together programs. However it suffers from a number of technical deficiencies when compared to other languages; particularly with regards to performance, multi-threading and memory.

Despite these problems, python is popular and provides a fast

Philosophically python is something of a mess. While C is quite clearly and imperative language, C++ and Java are object orientated, LISP is declarative, Python tries to be just about everything at once, and in each case mostly succeeds.

Python Interpreter

First, we will start by introducing the python interpreter. This program takes the python source code, compiles (on the fly) and then executes it.

```
python
```

However, we will want to use python version 3.6 or above, to check if the python command is using 2 or 3, run the python command with the –version flag.

```
python --version
```

If the output from the terminal is similar to

```
Python 2.7.14
```

Then the default python interpreter is version 2. In the event of this situation, the system may still have python 3 present but there will be a separate command called **python3**

```
python3
```

In this case you can set an alias in your `.bashrc` file to get the correct Python version. However, it is generally advised not to touch your system Python. This python installation may be used by some number of system packages, and these may depend on a particular version of Python.

Instead, a Python environment manager is typically used, either `py_env` or `miniconda`.

Unfortunately, each of these Python installations requires from 300MB to 2GB of disc space. As we can see, Python quickly gets expensive.

Dispatch Methods and Python

Python extends from last week's discussion of dispatch methods and structs. Imagine that each type in the language came with a builtin table of function pointers and associated dispatch methods. For example the `to_int` method might wrap all casts to integers.

```
struct py_int {
    int (*to_int)(void*);
    int val;
};

struct py_float {
    int (*to_int)(void*);
    float val;
}

struct py_str {
    int (*to_int)(void*);
    char* val;
}

int int_to_int(void* v)
{
    return ((struct py_int*)v)->val;
}

int float_to_int(void* v)
{
    return (int)(((struct py_float*)v)->val);
}
int str_to_int(void* v)
{
    return atoi(((struct py_int*)v)->val);
}

int to_int(void* v) // Dispatch method
{
    return ((int (*)(void*)(v))(v);
}
```

Here if each of the functions is set up correclty we can use the casting to ensure that the `to_int` function will return an integer for any of the above types.

Extending this to arbitrary functions only requires that we maintain the relative order of the function pointers - we could use a vtable in each case and ensure that the slots are ordered correctly, however this suffers from extensibility problems, namely that adding new elements to one object would require that we add them to all objects.

One workaround is to define a pair of associated arrays, one containing ‘keys’ the other function pointer values. In a simple example the keys would be the names of the functions, while the values were the function pointer lookups. More complex *hash* functions could reduce this from a string comparison to an injective function.

```c
struct associative_vtable
{
    void* (*vtable_lookup)(void*, char*); // Important first position
    char** keys;
    void* (**vals)(void*);
};

struct associative_vtable* constructor()
{
    static vtable_keys[] = {"to_int", "to_float", NULL}
    static vtable_vals[] = {int_to_int, int_to_float, NULL};

    static struct associative_vtable vt = {
        vtable_lookup,
        vtable_keys,
        vtable_vals
    };
    return &vt;
}

void* vtable_lookup(struct associative_vtable* vt, char* target_key)
{
    char** key = vt->keys;
    void** val = vt->vals;
    while (NULL != key)
    {
        if (0 == strcmp(*key, target_key))
        {
            return *val;
        }
        key += 1;
        val += 1;
    }
    // Not found
    return NULL;
}
```

```
// Dispatch Method
void* get_attribute(void* obj, char* attr)
{
    return (
    ((void* (*)(struct associative_vtable*, char*))obj)(obj, attr)
    );
}
```

With this model we can now do runtime checking of properties of an object. With dynamic memory we could extend those vtables at runtime. This is in effect a very general model of programming - if you can tolerate the overhead.

As it turns out this is the basis of python. The `dir` operator prints the list of keys associated with a type. All operators are syntactic sugar for dispatch methods that call elements of the vtables that implement the operation. These dispatch methods are in turn stored in `__builtins__` which is also a set of associative keys and function pointers. In fact all objects in Python are really just a pointer to an associative set of keys and pointers. To give an example:

```
x = 5 + 6
```

Is an expression in Python. This expression is also a string, so we can expand it. Builtins contains a dispatch method for the 'int' type.

```
x = int(5) + int(6)
```

But how did we know it was an integer? Builtins also contains a `type` operator.

```
x = type(5)(5) + type(6)(6)
```

Addition is relative to the integer operator, and `dir` on int shows us the `__add__` method.

```
x = type(5)(5).__add__(type(6)(6))
```

We're implicitly calling builtins here, let's make that explicit.

```
x = __builtins__.type(5)(5).__add__(__builtins__.type(6)(6))
```

We're not finished yet. Scope in python is also an associative set of keys (variable names) and values (pointers). So we need to define a scope. Builtins defines two scopes: `globals()` and `locals()` these do exactly what you think they might.

```
locals()['x'] = __builtins__.type(5)(5).__add__(__builtins__.type(6)(6))
```

Assignment is also an operator, `dir` shows us that it's the dynamic dispatch for the `__setitem__` method.

```
locals().__setitem__(
    'x',
    __builtins__.type(5)(5).__add__(__builtins__.type(6)(6))
    )
```

We're still not done - how does builtins know what `__setitem__` is? Well as it's also an associative array.

```
    locals().__getattribute__('__setitem__')(
    'x',
    __builtins__.type(5)(5).__add__(__builtins__.type(6)(6))
    )
```

Indeed we can extend this to all of our builtins calls, and to finding `__add__`.

```
    locals().__getattribute__('__setitem__')(
    'x',
    __builtins__.__getattribute__('type')(5)(5)
    .__getattribute__('__add__')(
        __builtins__.__getattribute__('type')(6)(6))
    )
```

And of course `locals()` and `globals` are also included in `__builtins__`.

```
    __builtins__.__getattribute__('locals')().__getattribute__(
    '__setitem__')(
    'x',
    __builtins__.__getattribute__('type')(5)(5)
    .__getattribute__('__add__')(
        __builtins__.__getattribute__('type')(6)(6))
    )
```

And that is how Python adds two numbers and saves them in a variable - all without specifying a single type. As variables are all just pointers, you can concieve of them as an array of `void*`s, and hence the type itself is irrelevant until we resolve the vtable.

Very importantly we can see how every operation resolves down to `builtins` and the `__getattribute__` function acting on strings.

We can colour our interpretation of this by also noting that globals and locals both contain pointers to builtins, so you can reinterpret this entire stack as calls to the local memory scope.

As we will see in future weeks, if all of Python is just associative arrays then we should be able to define our own, and just as importantly tweak elements of existing objects. Lastly we would expect that if we can copy an array then we can copy one of these associative arrays and add, remove or overwrite the copy with our own elements.

This control of associative arrays is the basis of both Python and object orientated programming.

Writing Python Source Files

Open your favourite text editor (atom, vim, emacs), create a file and save it as hello.py

After saving the file, we can started with writing a basic hello world program and introduce you to the print function.

```
print("Hello Python!!")
```

To run this program, we execute the python command with the source file name. As Python is interpreted, rather than compiled we don't need to compile it first.

```
python hello.py
```

Your program should output

```
Hello Python!
```

Unlike C, Python's print function works straight out of the box, and handles type conversion without the need for format strings.

```
print(5)
```

The print function acts on the `__repr__` or the `__str__` dispatch methods of a type to generate its string representation.

This genericises the print statement to all types in python.

Question 1: Echo

The input function in Python takes a string from standard input and saves it as a Python string type.

Write a Python program that takes input and prints it out again.

Python CLI

As python is interpreted, each line can be run independently. From this, we can use a use a python command line environment for testing Python snippets.

You can access the CLI using the `python` command.

Then you can install `ipython` and never use the regular CLI again.

`ipython` is a strict upgrade on the regular CLI, but comes with tab completion, colour completion, an inbuilt debugger and a large number of quality of life features.

You can install `ipython`, and other python packages using `pip`. You should check which python your pip points to, you can redirect your pip by loading it through the correct python with `python -m pip`. It might be an idea to alias this in your `.bashrc`.

Variables

Variables in Python are typed dynamically. That is that the type of the variable is assigned, and may be re-assigned based on the type of the right hand of the assignment.

The type of a variable can be determined using the `type` function.

```
x = 5
print(type(x))

x = "test"
print(type(x))

x = 3.5
print(type(x))
```

Comments

Once your program reaches a reasonable amount of complexity you should provide some kind of documentation so that you yourself can understand what you have written.

Code in python is typically commented using # or using multiline strings without assigning it to a variable.

# is typically used for inline and non-publishable documentation while multi-line strings are used for large code bases which will require developers to read about.

Multi-line strings can be delimited using triple quotes, either `'''` or `"""`.

Help and API Documentation

You will run into problems that will make you consult the documentation for the programming language. You will can access this resource by following this link: Python 3.6 Documentation

Or you can use the python interpreter's help function. This function will allow you to inquire about certain types and packages within the python ecosystem that you have installed.

For example, if we wanted to get help with `type`, we would write

```
help(int)
```

Or function

```
help(help)
```

or module

```
help(sys)
```

In ipython, the `?` unary operator acts as a shorthand for the help function.

Builtin Types

The python programming language has a number of built-in types that are considered *standard python* built-in types.

- Integer, **int** - Creates a integer from a series of numeric characters

  `45`

  `3`

  `90091`

- Floating point, **float** - Obvious difference is that floating point is denoted with decimal place

  `45.5`

  `2.2`

  `12 / 5`

- Boolean, **bool** - Boolean variables only have two values associated with them, `True` and `False`

  `a = True`

  `b = not a #This is False`

- String, **str** - When declaring a string within your code, it is denoted using the ' symbol or " symbol

  `"This is a string"`

  `'This is a string'`

  `This is not a string` and will cause a syntax error

Operators

During the lecture you will have been introduced to most of the operators used in Python. This list should provide a quick recap during the tasks of this tutorial and help you get familiar with them. You can find the full list of operations associated with a type using the `__dir__` builtin.

Arithmetic

- `+` : `__add__` Add operator, with numerical types (integers and floating point) numbers this adds two numbers while with Strings it will concatenate both strings and create a new string.
- `-` : `__sub__` Subtract operator which will subtract the number from the right hand side from the left hand side.
- `*` : `__mul__` Multiplication operator, allows two numerical types to be multiplied, this can be used with string types where it will multiply contents of the string N times and produce a new string.

- `/` : `__truediv__` Division operator, allows two numbers division of two numbers, division by 0 will result in a 'ZeroDivisionError'
- // : `__floordiv__` Floor division operator, divides and takes the floor of the result, casting to int.
- % : `__mod__` Floor division operator, divides and takes the floor of the result, casting to int.
- (« » : `__lshift__`, `__rshift__`. Left and right shifts.

As we have the ability to keep adding methods to an object's associative array without binding an operator we also get `__abs__` `__ceil__` `__floor__` and others. Find them by using `dir`.

Comparative

- `==` `__eq__` Equal to, checks for equality between two values
- `<` `__lt__` Less than, checks a value is less than another value
- `<=` `__le__` Less than or equal to, checks a value is less than or equal to another value
- `>` `__gt__` Greater than, checks a value is greater than another value
- `>=` `__ge__` Greater than or equal to, checks a value is greater than or equal to another value
- `!=` `__neq__` Not equal to, checks for inequality between two values
- `is` `__is__` Is equal to - checks that the address of the two objects are identical using the `id` operator.

Logical

- `not` Negation, returns a boolean value (True or False) depending on the value. ie if a is true, then **not** a is false
- `and` Logitcal AND, returns True if both values (lhs and rhs) are true, otherwise False
- `or` Logitcal OR, returns True if at least one values (lhs and rhs) are true, otherwise False

Bitwise

- `&` Binary AND operator which will return the value of the bits that only exist in both numbers.
- `|` Binary OR operator, will return the value of the bits
- ˆ Binary XOR operator, will return a value where bits exist in both values but if the same bit exists in both values it will exclude it.
- Binary operator, will flip the bits of a value
- « Left shift, will shift all bits left by N
- » Right shift, will shift all bits right by N

Question 2: Assignment

Take `hex` of the output of `id` for an object in Python.

- What is the `id` of an object?
- What does the following code imply about how Python assigns variables. How does this differ from C?

```
x = 5
print(hex(id(x)))
x = 6000
 print(hex(id(x)))
```

- What does the following code imply about how Python allocates integers?

```
x = 5
y = 5
x += 1
y += 1
print(id(x) == id(y))
```

- What about if I do the same thing with some different values? Have your beliefs in how Python implements integers changed?

```
x = 256
y = 256
x += 1
y += 1
print(id(x) == id(y))
```

If you believed that Python performed constant memory allocations for integer constants, then how could you find this to be obviously wrong for small amounts of memory?

Question 3

*[Python Integers]

- `bin` claims to show a binary cast of integers in Python. How does `bin` of negative numbers differ from C's implementation. Given integers are implemented in C is the bin function truthful?
- What is the sizeof an integer in Python? Hence what is the maximum size of an integer in Python? (Hint, try allocating two variables in order and use the offset between them to approximate the amount of memory required, alternatively use `sys.getsizeof`.

- What does this imply about the complexity of the implementation of mathematical operations in Python?
- Does Python perform type promotions in the same fashion as C?

Question 4: Naming conventions

In the future, please refer to the PEP8 style guide. Specifically: PEP-8 which outlines naming conventions for variables and functions.

In this exercise, there is a table that lists variable names on the left most column and your task is to determine if a variable is:

- Valid variable name
- Adheres to PEP8 Convention (mainly snake_case)

`age`	
`1st_element`	
`no_times`	
`SIZE_OF_FIELD`	
`_42`	
`darknessAllAround`	
`superdupervariable`	
`coolplayer2`	
`UserName`	
`_var1_`	

Python Strings

Strings in Python are tuple backed rather than the array backed strings in C. This means that they are dynamically sized, but are immutable.

As a result, whenever we wish to modify a string, we must copy it to a new variable and perform the modifications before assignment.

Strings also have a few special operators, we'll be paying a bit of attention to addition, which concatenates two strings, and multiplication by an integer, which duplicates a string that many times.

Other methods for strings can be seen using `dir(str)`.

Formatting output

There are quite a few different ways of formatting strings in Python. Ignoring the format string type, here are a few of them.

```python
age = 20
print("I am " + age + " years old!")
print("I am ", age, " years old!")
```

Instead we can create placeholders within a string that can be parsed and evaluated.

```
print("I am {} years old!".format(age))
print("I am %d years old!" % (age))
```

This is a lot more robust and flexible then continually concatenating strings and different data types together to construct a string.

Using C style format strings

Python supports two types of formatting of strings, the old way is using a C-like printf style of formatting strings. The string contains datatype specific placeholders that values are inserted into.

Specifier	Meaning	Options
%d	Signed integer decimal	%4d (padding)
%f	floating point decimaly	%.2f 2 characters after decimal point
%s	String	%-10s, %.5s truncation
%c	Single character	%02 2 character padding

We are not limited to strictly specifying data but we can specify how that data is presented as well, either through truncation, padding or constraining the decimal place on floating point numbers.

```
s = 'Pirates! The Game'
prog = 1/3
times = 4100202

print("Getting: %s, Progress: %.2f, Downloaded: %8d" % (s, prog, times))
```

You may refer to this part of the documentation as how printf-style string formatting works.

printf-style String Formatting

Using format functions (the better way!)

In later versions of python 2, a new way of formatting strings was available which is a lot more flexible and easier to perform. You can still use the classic C-style printf method but we also have the power of a new templating system for data formatting in strings.

Specifier	Meaning	Options
{}	Data placement	.format('data')
{named}	Named data placement	.format(named='data')
{index}	Index data placement	'{1} {0}'.format('one', 'two')
{data[0]}	named data with index or key	'data[0].format(data=d)'

```
s = 'Pirates! The Game'
p = 1/3
t = 4100202

print("Getting: {}, Progress: {:.2f}, Downloaded: {:8}".format(s, p, t))
```

You may refer to Custom String Formatting from the python 3 documentation.

Both styles are very well compared and shown on PyFormat which shows different usage and formats between strings.

Question 5: Boolean expressions

Python comes with explicit booleans, along with a more verbose implementation of boolean logic.

Given the following statements

```
a = True
b = False
x = 50
```

Evaluate the following expressions

`a and b`	______________________
`not(not(a)) or b`	______________________
`a and not b or not a`	______________________
`(b and not(b)) or (a or not(a))`	______________________

Question 6: Variable assignment and computation

Given the following program. Annotate each line and show the value of each variable.

```
a = 10
b = 0
b = a
a += a + b
x = a - b
x = a * b
x -= a * b
a = a/2
```

Question 7: State of execution of a program

Given the following python program. Annotate each line with the data type and the current value the variable contains.

```
a = 3
b = 4
a = '3'
j = a + str(b)
b += 0.5
o = j
o = len(j) < 2
t = str(o) + j
```

Discuss with your tutorial group how python would handle changing the data type of a variable? Think about how the interpreter can handle this case and what advantages and disadvantages when programming.

How could we test what the type a variable is and why would we want to do that?

Question 8: Capitalise it all!

Given any line of input to your python program, capitalize all the letters of the string and output it.

```
> python cap.py
All that glitters is not gold
ALL THAT GLITTERS IS NOT GOLD
```

Question 9: Celsius to Fahrenheit

Write a program that will take celsius as input and convert it to fahrenheit. The output of fahrenheit should only show up to 2 decimal places.

```
python ctof.py
Degrees in celsius: 2.25
Conversion to fahrenheit is: 36.05
```

To convert from celsius to fahrenheit, you are given the formula:

$F = C * 9/5 + 32$

Question 10: Fahrenheit to Celsius

After finishing the celsius to fahrenheit conversion, write a program that will calculate fahrenheit to celsius.

Extension

Question 11: Matches

You are to write a program that will calculate the total number of sets a tennis player has won. You will be provided 2 player names and 5 sets of the games won by each player. If a player has a set where they have won 6 games, they will have won that set, else they have lost.

```
python set.py
Please enter <player1> <player2> <scores>
James Jim 6-5 5-6 2-6 6-4 5-6
James won 2 sets and 24 games
Jim won 3 sets and 27 games
```

Question 12: RGB 24bit Colour

Colour is typically stored as an integer when we talk about monitors, as a programmer, we need to be able to extract and change colour values that are stored as an integer. Bitwise operators such as bit-masking and bit-shifting make this very easy.

How an RGB colour value is stored: Red: First 8 bits Green: 8 bits after Blue: Last 8 bits

Each primary extracted will be within the range of 0-255. You are to also read the number in using hex, a hex value of 0xFFFFFF will correspond to White (255, 255, 255).