Introduction to Python

ESR Programming Workshop - IGNITE

Killian Smith

Winter 2018-19

Python

Overview

- First released in 1991 by Guido van Rossum.
- Interpreted general purpose language.
- Scope is defined by whitespace, not brackets.
- Has so called *"duck typing"*.
- Tries to be Procedural, Object Oriented, and Functional.

What is it good for?

- Teaching programming
  - Rich libraries and a large community
  - Fewer special symbols than some languages
  - No explicit typing
- Writing scripts or as a high-level wrapper for other programs.
- Applications where portability is desired over performance.

What is it *not* good for?

- Computationally intensive computing.
- Large standalone applications.
- A replacement for a compiled language.

Outline

- Briefly touch on the history of programming languages.
- Talk about **Object Oriented** programming.
- Review Python syntax.
- Review UML diagrams.
- Build a Linked List.

A Brief History of Programming

Machines (at the moment) can only understand groups of binary digits, called a machine *word*, arranged in a sequence.

- Humans are absolutely terrible at reading or writing these sequences.
- So we made bijective acronyms for pieces of these *words*, called assembly language.
    - `bne $r1, $r8, label`
    - `add $r1, $r0, $r1`

Abstraction

This *assembly language* was a step in the right direction, but. . .

- It is dependent on the chip architecture.
- There is excessive *boiler plate* code.
- The language does not promote code reuse.
- Many simple abstractions like contiguous memory blocks or conditional statements are not available.

To solve these issues, the first programming languages were made.

Procedural Languages

This family of languages is closely linked to assembly language, and languages like `C` are jokingly referred to as high level assembly language.

- The programs are still written sequentially.
- The programs declare variables and mutate them during runtime.
- Languages of this paradigm are often called *imperative* languages since they tell the machine step by step how to execute.

Message Passing

Procedural languages do a good job at abstracting control structures, but still do a poor job at abstracting away state. The *message passing* paradigm was created to solve this.

- Every unique structure is self contained, and are called *objects*.
- These *objects* interact by sending messages and changing variable states. In this way, state is isolated within an object.
- In nearly all cases, object internals are written in a procedural style. This combination of message passing and procedural code is called *Object Oriented* programming, or *OO* for short.
- This is roughly where Python falls (or at least it is best practice).

Functional Languages (side note)

Around the same time as the first procedural languages, an interpreter for the mathematical model of computation λ-calculus was created LISP.

- This mathematical approach to computation is called *declarative* as it tells the computer what to do, not how to do it.
- These languages have historically not been popular do to poor performance, but advances in compilers, garbage collection, concurrency, and CPUs have made them just as fast, and sometimes faster, than highly optimized imperative programs.
- If you can, **learn a functional language**. These types of languages are the future of programming. Every day, concepts from functional languages are incorporated into mainstream languages.

Logic Languages (side note)

The *logic* paradigm of programming is much lesser known than its relatives. In the early days of AI it was a popular avenue of research, but it never fully recovered after the AI winter.

- This paradigm mimics logic. By following logical inference rules, these languages can solve for constraints within a program expression.
- A classical example is the Prolog family of languages.
- Although no longer widely in use, concepts from logic languages are used in language *type systems*.

OOP in Python

Objects are encapsulated instances of structure, and have two components:

- The encapsulated statefull variables of the object, called the *constructor*.
- Set of methods to interact with the rest of the universe.

Class in Python

Python calls the template for an object a class. A class starts with the `class` keyword and a class name, and all preceding indented lines are considered a part of the *class body*. This class can also optionally inherit from a super class.

```
class name(opt_super):
    ...
```

Constructors in Python

The constructor for a class has a special syntax, and should be the first block in the class.

```
def __init__ (self, arg1, arg2, ...):
    self.var1 = ...
    self.var2 = ...
    ...
```

Methods in Python

Methods are declared with the `def` keyword, a method name, and a comma separated argument list. All preceding indented lines are considered a part of the *method body*.

```
def name (self, arg1 , arg2, ...):
    ...
```

Reserved Methods in Python

There are some special methods in Python that are used to automatically overload some existing functions. These appear as

```
def __keyword__(self, ...):
    ...
```

declarations. The full list of special methods can be seen in the language documentation.

Reserved Methods in Python (cont)

In our lab, the only special method that we are using is the `__str__` method. This method defines how to display our class as a string. This overrides the `str()` Python function.

```
class Point():

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "(" + str(self.x) +
               "," + str(self.y) +
               ")"

    ...
```

Imperative Python

In addition to the object level features, Python also has was to manipulate data within an object.

- These operations are *usually* imperative and *usually* mutate the structures...
- The only real convention in Python is that there is no convention...

Branching in Python

Conditional branching is achieved using `if` statements in various forms.

Form 1:

```
...
if <predicate>:
    <body>
...
```

Branching (cont)

Form 2:

```
...
if <predicate>:
    <body1>
else:
    <body2>
...
```

Branching (cont)

Form 3:

```
...
if <predicate_1>:
    <body1>
elif <predicate_2>:
    <body2>
...
elif <predicate_n>:
    <body2>
else:
    <body2>
...
```

Branching (cont)

Note that the only *needed* branching syntax is form 2. The other syntactic forms can easily be written in terms of `if-else`.

- In form 1, the `else` is implicit.
- In form 3, the `elif` blocks can be written with an `else` followed by a nested `if`.

Branching Example

```
x = 42

if x > 0:
    print(‘x is positive’)
elif x < 0:
    print(‘x is negative’)
else:
    print(‘x is zero’)
```

Evaluates as. . .

Branching Example

```
x = 42

if x > 0:
    print('x is positive')
elif x < 0:
    print('x is negative')
else:
    print('x is zero')
```

Evaluates as. . .

```
> ''x is positive''
```

Looping in Python

Python provides two primary ways to loop:

- `while` loops
- `for` loops

While Loops

The simplest possible looping construct, comprising of 2 components:

- A predicate, meaning an expression that strictly evaluates to a boolean value.
- The body of the loop that is executed.

```
...
while <predicate> :
    <body>
...
```

While Loop Example

```
i = 1

while i < 5:
    print(i)
    i = i + 1
```

Evaluates as. . .

While Loop Example

```python
i = 1

while i < 5:
    print(i)
    i = i + 1
```

Evaluates as. . .

```
> 1234
```

For Loops

A common use for loops in imperative languages is iterating over collections. Python has a special syntax for doing this for many types of built in sequences, the `for` loop. The syntax for this loop is:

```
for <var> in <sequence>:
    <body>
```

For Loop Example

```python
for i in [1,2,3,4]:
    print(i)
```

Evaluates as. . .

For Loop Example

```
for i in [1,2,3,4]:
    print(i)
```

Evaluates as. . .

```
> 1234
```

Recursion in Python

WARNING - While Python does support *recursion*, it does not support *tail call optimization*. **Do not** use recursion on large data structures. Your programs *will* blow up the stack.

If you do want to use recursion on smaller data structures, it is as simple as calling the same function within that functions body.

Recursion Example

```
def foo(x):
    if i >= 4:
        print(x)
    else:
        print(x)
        foo(x+1)

foo(1)
```

Evaluates as. . .

Recursion Example

```python
def foo(x):
    if i >= 4:
        print(x)
    else:
        print(x)
        foo(x+1)

foo(1)
```

Evaluates as. . .

```
> 1234
```

Linked List

One of the simplest structures that we can make is a *linked list*.

$$L_a = 1 + a \times L_a$$

This means we can either have some *unit* type, in this case an empty list, or we can have some content a followed by the remainder of the list.

Linked List (cont)

A linked list can be decomposed using the operations *head* or *tail*.

Head

The head operation takes the first element of the list, taking $O(1)$ time.

Tail

The tail operation returns the sublist without the first element, taking $O(1)$ time.

Operations

The following operations are typically supported when working with singly linked lists:

- A predicate for the empty list
- A lookup function
- An insertion function
- A deletion function

List Data Structure

```
class LinkList:

  def __init__(self, val):
    self.val   = val
    self.tail  = None

  ...
```

Display

```
...
def __str__(self):
  res = ''
  if not self is None and not self.tail is None:
    res += str(self.val) + ' ' + str(self.tail)
  return res
...
```

Insertion

```
...
def insert(self, val):
  if self.val is None:
    self.val = val
  else:
    tmp = LinkList(self.val)
    tmp.tail = self.tail
    self.val = val
    self.tail = tmp
...
```

Search

```
...
def find(self, val):
  if val == self.val:
    return True
  elif self.tail != None:
     return self.tail.find(val)
  else:
     return False
...
```

Deletion

```
...
def delete(self, val):
  parent  = None
  current = self

  while current:
    if current.val == val:
      if parent:
        parent.tail = current.tail
      else:
        self = current.tail

    parent  = current
    current = current.tail
...
```

Main Method

```python
def main():
    root = LinkList(5)
    root.insert(2)
    root.insert(8)
    root.insert(3)
    root.insert(1)
    root.insert(6)
    root.insert(10)
    ...
```

Main Method (cont)

```
def main():
    ...
    print(str(root))
    print(str(root.find(1)))
    print(str(root.find(7)))
    root.delete(1)
    root.delete(2)
    print(str(root))
```

Call Program

We also need to tell Python how to enter the program when called as an executable.

```
if __name__ == "__main__":
    main()
```

Thanks for Listening

Questions or Comments?