Python

- What is *? In Regular Expression?
- Parameterization and string substitution
- All About Importing Modules and from Packages
- Global variables with imports
- Unpacking operator in code and function header
- Python Decorator Demystify

What is *? In Regular Expression?

https://stackoverflow.com/questions/3075130/what-is-the-difference-between-and-regular-expressions

Parameterization and string substitution

String substitution

In the context of building a database query like so:

`CREATE TABLE fish (name TEXT, species TEXT, tank_number INTEGER)` you would likely want to include for example user input for say a search query into a database. There are two ways to go about it and the first way to build a query is via string substitution and this is the bad way.

Say the user input is in the variable `name` and the user can put whatever name they like including special characters. We have our search query to find the particular `name` that the user inputted as follows:

```
query = f"SELECT * from user_table WHERE name='{name}'"
```

Using f-string we are interpolation the variable with the query and this give the chance for SQL Injection attack to occur. This is because the user can specify something like `name = "Tom' OR 1=1"` as it's input and when the variable is used for interpolation for building the query it results into:

```
name = "Tom' OR 1 = 1"

"SELECT * from user_table WHERE name='{name}'"

"SELECT * from user_table WHERE name='Tom' OR 1 = 1"
```

And when the query is executed it will pick everyone from the database and retrieve their information (possibly private information).

Parameterized queries

Now for a much safer approach to building query is via parameterized queries. In this case, the variables that are used to build the query are pass as parameters and not used directly in a string interpolation. How you do this will depend on the library that you use, for example, in Python when you execute a query you can pass a parameterized string along with the parameters to build a

parameterized queries.

The idea is that if the user input are passed as parameters, they no longer have the chance to mess with the query since there are no interpolation, the user input is not used to build a query but rather as a parameter. **The variables themselves will no long be used as part of an executable code but rather treated as literal values.**

```
query = "SELECT * from user_table WHERE name=?"
params = (name)
cursor.execute(query, params)
```

Now no matter what the user input, even `Tom' OR 1 = 1` as `name` it will be treated literally, as you are looking for a person named `Tom' OR 1 = 1` in the database.

Further clarification

In the future, if you are back to this post and wonder isn't this the same as string substitution? You would be wrong.

The way that parameterized query works is that the query is first sent to the SQL engine, and the database will know exactly what this query will do, and only then it will insert the username and passwords as LITERAL VALUES. So the user input values cannot affect the query in anyway.

It is separating the values with the queries, the query is first sent to the database and database will know what it does, then it will ask for the parameters that you have layed out and use it LITERALLY, not as part of the query anymore.

In the previous example, you are asking the database can you select all the columns from `user_table` where the name is equal to a parameter, the database understood your request, then ask you for the `name` and the name is provided as parameter. Even if you sent `Tom' OR 1 = 1` the database will be looking for a person named `Tom' OR 1 = 1`.

All About Importing Modules and from Packages

Module?

A module is just a Python file with the corresponding `.py` extension. So if you're talking about the math module then there is a corresponding `math.py` file that contains functions, classes, and constants that are meant to be used by other Python files.

Where does Python look for modules

If you have written a python module under say `a_module.py` in a directory of `code`.

And you have a script called `a_script.py` in directory called `scripts`. You would like to use the `a_module` in `a_script.py` by importing it.

```
import a_module
```

Then you try to run the `a_script.py` by running `python3 scripts/a_script.py` it will fail with

```
$ python3 scripts/a_script.py
Traceback (most recent call last):
  File "scripts/a_script.py", line 1, in <module>
    import a_module
ModuleNotFoundError: No module named 'a_module'
```

When Python imports a module it will try to find a package or module. But where does it look? Python has a simple algorithm for finding a module with a given name. It will look for a file called `a_module.py` in the directories listed in the variable `sys.path`.

```
>>> import sys
>>> type(sys.path)
<class 'list'>
>>> for path in sys.path:
...     print(path)
```

```
...
/Users/brettmz-admin/dev_trees/psych-214-fall-2016/sphinxext
/usr/local/Cellar/python/3.7.2_1/Frameworks/Python.framework/Versions/3.7/lib/python37.zip
/usr/local/Cellar/python/3.7.2_1/Frameworks/Python.framework/Versions/3.7/lib/python3.7
/usr/local/Cellar/python/3.7.2_1/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload
/Users/brettmz-admin/Library/Python/3.7/lib/python/site-packages
/Users/brettmz-admin/dev_trees/grin
/Users/brettmz-admin/dev_trees/rmdex
/usr/local/lib/python3.7/site-packages
```

It doesn't search recursively, it will only search in the directory that is listed under `sys.path`. And as you can see the `code` directory is not in `sys.path` which is why Python could not import `a_module.py`.

To fix this you can just simply append to the `sys.path` list like so:

```
import sys
sys.path.append('code')

import a_module
```

Now this will work as expected. This simple search algorithm for module also works for packages, it searches for packages then the module the same way.

> Information from: https://bic-berkeley.github.io/psych-214-fall-2016/sys_path.html

What is Namespace package and Regular Package

A namespace packages are special packages that allows you to unify two packages with the same name but are at different directories:

```
path1
+--namespace
   +--module1.py
   +--module2.py
path2
+--namespace
```

```
    +--module3.py
    +--module4.py
```

Notice that in order to make Namespace package work you would have to add the two paths `path1` and `path2` to `sys.path`. Then you can import the four modules by doing

```
from namespace import module1
from namespace import module2
from namespace import module3
from namespace import module4
```

Or import specific functions from the module

```
from namespace.module1 import boo
```

Namespace packages basically unifies the two packages with the same name in a single namespace. You can import all four modules freely.

However, if either one of the `namespace` packages gain an `__init__.py` then it will become a normal package, and the unification is no longer as the other directory will be ignored.

If both have `__init__.py` the first one encountered in `sys.path` is the one being used.

Regular package

A regular package is a collection of modules. In order to make Python recognize that it is a regular package you must add `__init__.py`. Without `__init__.py` then it will be interpreted as a namespace package which doesn't fit 99% of normal use cases.

So the moral of the story is that if you're creating packages, just put `__init__.py` in it unless you have the special use case of needing to unify two namespace packages that are in different directory.

Regular packages are also searched in `sys.path` just like everything else.

Global variables with imports

Sharing global variables between modules

If for whatever reason you need to share global variables you need to keep in mind that global variables in Python are global to a module, not across modules. Which is different than say `static` variable in C.

To make your global variable be able to be shared cross module you would obviously declare a global variable in a module, and the best practice is to declare it in a dedicated module file called `config.py` and have everyone that needs that global variable import it into their scope.

```
# config.py
a = 69 # Variable declared on top level are global by default
```

```
# cool.py
import config
print(config.a)
```

> A word of warning, don't use `from` import unless you want the global variable to be a constant. If you do `from config import a` this would create a new `a` variable that is initialized to whatever `config.a` is at the time of the import, and this new `a` variable would be unaffected by any assignments to `config.a`. So it is not global global anymore.

Unpacking operator in code and function header

Unpacking

In Python you are allowed to do deconstruction similar to how you can do deconstruction assignment in JavaScript. You can take a list and then assign each of the elements individually to variables on the left hand. For example:

```
(a, b, c) = (1, 2, 3)
# a = 1
# b = 2
# c = 3
```

If you are using this simple unpacking assignment then the values on the right hand side will be automatically assigned to the variables to the left according to their position in the tuple.

When you unpack values into variables the number of variables present must be exactly equal to the number of elements in the tuple, else ValueError

You can use this tuple unpacking to unpack a String into characters, unpack list into variables, unpack a generator, and unpack a dictionary as well.

The left hand side can be a list as well but is rarely used.

* Unpacking operator

The * operator extends the unpacking tuple functionality to allow you to pack multiple values into a single variable as a list. For example:

```
*a, = [1, 2]
# a = [1, 2]
```

In this example, the variable a gets stuffed two values 1 and 2 into it as a list. Notice the usage of comma there, it is required otherwise it will be a normal list assignment statement and not unpacking.

You can have as many variables as you want on the left hand side, but there can only be one starred variable. The starred variable contains all the rest of the values that are not distributed to the mandatory variable in the list, even if it has no value. If there are no element left to distribute to the starred variable, then the starred variable will just be an empty list:

```
a, b, *c = [1, 2]
# a = 1
# b = 2
# c = []
```

Function header with *args and **kwargs

Sometimes in Python function header you will see functions that takes in variable number of position arguments with *, and variable number of keyword arguments with **. For example:

```
def func(required, *args, **kwargs)
```

`args` and `kwargs` are both optional, and are automatically default to `() and {}` respectively, if none are provided for the function.

Calling the function above would be like so:

```
func("hello", 1, 2, 3, site="hi", oh="lord")
# required = "hello"
# args = (1, 2, 3)
# kwargs = {"site": "hi", "oh": "lord"}
```

Side note on method headers

1. Variables without default value must be before those that have default values

   ```
   def test_func(a=10, b, *args, **kwargs)
   ```

   Is not valid Python function
2. `*args` must be fore `**kwargs`. They cannot switch, otherwise, invalid Python syntax
3. Follow the structure of `func`, don't try to put position arguments after `*args`.

Python Decorator Demystify

What are Decorators

First and foremost, a decorator is a function that takes in a function and return another, letting you add extra behavior without changing the original function's source code.

```
def decorator(fn):
  def wrapper(*arg, **kwargs):
    print("Before")
    ret = fn(arg, kwargs)
    print("After")
    return ret
  return fn
```

This is a simple decorator that allows you to surround the execution of your function with a before and after print statement.

> It is important to note that, if you're not planning on modifying or extend any ability for a decorator, you can actually just return the function, without writing that wrapper function. You only need to write the wrapper function if you **need to extend or add onto what the original function is doing.** This will be important later.

You can then use the decorator by using it with a function like such:

```
@decorator
def add(a, b):
  return a + b
```

When you call `add(1, 2)` it will print "Before" and "After" and then returns you the value of 3 from the function.

What Python is essentially doing underneath is something like this:

```
add = decorator(add)
```

It calls the decorator function with your add function as it's argument and reassign it to add. Then the actual function that gets called is wrapper.

Decorator with Arguments

If you want to pass arguments to your decorator as to customize the decorator further, you can! However, you will need more layer of nesting:

```
def decorator_with_arg(arg):
  def decorate(fn):
    def wrapper(*arg, **kwargs):
      print("before is called with", arg)
      ret = fn(arg, kwargs)
      print("after is called with", arg)
      return ret
    return wrapper
  return decorate
```

Now you can see this is a little bit more complicated, if you want your decorator to have inputs from the user, then you will need to change the arguments for the most upper level function to take in what you want. In this case, it is just `arg` which can be anything since we are just printing it out.

But inside you can see that it is the decorator! So the most outer layer is returning decorate, which we have defined previously to be a function that takes in a function and returns another extending it's capability. You can see that it is another layer of nesting!

```
@decorator_with_arg("jessie")
def add(a, b):
  return a + b
```

If you decorate your function with it and call it it will print "before is called with jessie" and "after is called with jessie" and return you the value of a + b.

Functionality Python is doing something like this:

```
add = decorator_with_arg("jessie")(add)
```

The first function call results in a decorator being resulted, and it is then called with your function as it's parameter to decorate.

What if you need argument, but don't want to modify the function

If all you want to do is pass argument to the decorator, i.e. for logging purposes, you actually can just shorten it to 2 level of nesting, you do not need three.

```
def decorator_with_arg(arg):
  def decorator(fn):
    print("Inside the decorator", arg)
    return fn
  return decorator
```

So same thing if you call it with:

```
@decorator_with_arg("mark")
def add(a, b):
  return a + b


// underneath
add = decorator_with_arg("mark")(add)
```

The decorated function will function the same as it was before, BUT when it was initially loaded by python and decorated during runtime, it will print "Inside the decorator mark". But since the actual function itself is not changed, it is just returned as it is (no wrapper function within it, the same fn is returned as it is). This can help you achieve with meta logging, or registering of functions without changing the functionality of the original method.