Run Down on ASTs

Author: Keenan Cunningham

Role: Data Science Intern

Date: 09-30-2024

Python's `ast` module can be daunting to look at. This is especially true when the program you wish to generate using the `ast` module is complex. But remember, computers are driven by logic. They are deterministic. Therefore, in this sprawling mess, there is some order. Once we establish and learn that order, through theory and practice, we will demystify the `ast` and be able to use it with competence and confidence.

Throughout this file, you will incrementally build a program using the `ast` module. We will start with the largest component of the `ast` module -the `ast.Module` class- and work our way down toward finer and finer detail. By the end of this reading the goal is to have a solid grasp of the `ast` module and to establish that we can rewrite the following program in the form of an **abstract syntax tree**.

```
class ShoppingList():
    def __init__(self, item_list: list[str] = None):
        self.item_list: list[str] = item_list

    def next_item(self) -> str:
        if (self.item_list == []):
            return None
        else:
            return self.item_list[-1]  # returns the last item in the list

    def add_item(self, item) -> None:
        self.item_list.append(item)

    def add_items(self, items: list[str]) -> None:
        self.item_list.extend(items)

    def remove_last_item(self) -> str:
        if self.item_list == []:
            return None
        else:
            return self.item_list.pop()


def does_list_contain(item: str, list: list[str]) -> None:
```

```
    for list_item in list:
        if item == list_item:
            return True
    return False


# -- MAIN --
# create a shopping list and pass its constructor a list containing "apples"
my_item_list = ["apples"]
my_shopping_list = ShoppingList(my_item_list)

# add the items "eggs" and then add the items "ham" and "cheese" at once
new_item = "eggs"
my_shopping_list.add_item(new_item)

new_items = ["ham", "cheese"]
my_shopping_list.add_items(new_items)

# remove the last item from the shopping list and print it
removed_item = my_shopping_list.remove_last_item()
print(removed_item)

# print the final list
print(my_shopping_list.item_list)
```

Starting at the Top: The `ast.Module` Class

At the top of every *abstract syntax tree* that we build will be an `ast.Module`. The `ast.Module` class can be thought of as *the file that we are writing to*. It's an empty Python file that has no lines of code in it.

The `ast.Module` class has only one part, the `body`. The `body` is a list which will hold every single `ast` component we build from here forth.

The simplest `ast.Module` that we can make would be defined: `ast.Module(body=[])`. If we were to call the function `astor.to_source` on this, we would receive nothing. This is because we haven't added anything to the `body` of the `ast.Module`; or equivalently, we have not *written any code to our python file*.

```
# copy this into a python file and try it yourself
import ast
import astor
```

```
module = ast.Module(body=[])    # simplest module we can make
print(astor.to_source(module))  # converting it to source code
```

Adding Code with `ast.Assign`, `ast.Name`, and `ast.Constant`

Now lets add something to our file. A natural next step is learning to assign a value to a variable. Let's *assign* the *name* `x` to the *constant* `1`. The resulting code generation will be `x = 1`.

`ast.Constant`

In order to generate this, we need to represent each part using the `ast` module. We will begin by representing the constant `1`, using the `ast.Constant` class. This class takes exactly one parameter, the `value` that we want to represent.

```
ast.constant(value=1)
```

`ast.Name`

Next, we need to give a name to the memory location that we want to assign the constant `1`. In the language of Python ASTs, this is the task of the `ast.Name` class. This class has two parameters: `id` and `ctx` which stands for *"context"*.

The `id` will be the **variable** name as a **string**. Since we want to assign a value to a variable `x`, our `id` will be `"x"`. Remember its a variable name as a string and not just a variable. The context will be one of three values:

- `ast.Load()` - for reading a variable
- `ast.Store()` - for assigning a value to a variable
- `ast.Del()` - for deleting a variable

In our case, we are assigning a value to our variable, so we will assign `Store()` to the `ctx` variable. Thus to declare a name that we want to assign a variable to, we write

```
ast.Name(id="x", ctx=Store())
```

`ast.Assign`

Now, we are ready to assign the constant number `1` to the variable `x`. To make this assignment, we will use `ast.Assign` which takes three parameters: `targets`, `value`, and `type_comment`.

1. `targets`: A list containing the thing or things that you want to assign a value to. If you list multiple targets, you will be assigning the same value to all of them.
2. `value`: Any constant that you want to assign to your target. This can be a constant number such as `1`, a string, or the value `None`.
3. `type_comment`: an **optional** string with the type annotation as a comment. We will omit this for now.
   Before we attempt to write this statement to our module, lets look at the anatomy of an `ast.Assign` to help familiarize ourself with this common AST structure.

```
ast.Assign(
        targets=[],
        value= #something
)
```

> Often we see the AST classes spanning multiple lines. This is a stylistic choice which may make things easier to read. This statement is exactly the same as `ast.Assign(targets=[], value=)`

Now, lets give this instance of the `ast.Assign()` class a target and a value, our `x` and `1`; respectively.

```
module = ast.Module(body=[])

assignment = ast.Assign(
        targets=[ast.Name(id="x", ctx=ast.Store())],
        value=ast.Constant(value=1)
)

module.append(ast.assign)
```

Notice that we created a variable called assignment and stored a small *abstract syntax tree* inside of it, our `x = 1` assignment. We can now append that to the body of module. This is totally acceptable and equivalent to:

```
# copy this into a python file and try it yourself
import ast
import astor

module = ast.Module(
        body=[
                ast.Assign(
                        targets=[ast.Name(id="x", ctx=ast.Store())],
```

```
                        value=ast.Constant(value=1)
                )
        ]
)

print(astor.to_source(module))  # converting it to source code
```

We are welcome to add items to the body of module -*write lines of code to the python file*- using either technique. This is a style choice. It is also common to mix up the styles; using the one that you believe is best for the current object of interest.

Exercise:

In the program at the top, we assign the variable `new_item` to the string constant: `"eggs"`. Make an AST that does this and print it to the console.

Other Constants and Datatypes

Usually when we think of constants, we restrain our thought to numeric values. But the `ast` module doesn't. A constant can be a number, string or the value `None`. For other values, the `ast` module defines the datatypes: string, formatted-string, list, dictionary, set, and, tuple with their very own node.

After completing the exercise, you should have the line `new_item = "eggs"` printing to the console. In the main section of the above code we make two more variable assignments both of which consist of assigning a list of strings. The only piece of information that will be new, is the creation of a list.

We can define a list literal using the `ast.List()` class. The class has two parameters: `elts` and `ctx`. We are familiar with `ctx` and know the values that it can take. The `elts` parameter is new and stands for *elements*. The `elts` parameter takes a list of comma-separated-constants. We can construct the heterogenous list `[1, "two", 3.0, 'four']` by typing

```
ast.List(
        elts=[
                ast.Constant(1),
                ast.Constant("two"),
                ast.Constant(3.0),
                ast.Constant('four')
        ],
        ctx=ast.Load()
)
```

Moreover, we could assign it to a variable called `my_list`, within a module by typing

```
ast.Module(
        body=[
                ast.Assign(
                        targets=[ast.Name(id="my_list", ctx=ast.Store())],
                        value=ast.List(
                                elts=[
                                        ast.Constant(1),
                                        ast.Constant("two"),
                                        ast.Constant(3.0),
                                        ast.Constant('four')
                                ],
                                ctx=ast.Load()
                        )
                )
        ]
)
```

> Here we use `ctx=Load()` because we are loading this list into the program. Then we use `ctx.Store` when storing this loaded value to the variable `my_list` using `ast.Assign`.

Exercise:

In addition to the assignment that you made in the last exercise, write the assignments:

- `my_item_list = ["apples"]`
- `new_items = ["ham", "cheese"]`

Be careful to ensure that each assignment occurs in the same order that they are in the program at the top. When we complete every exercise, we will have created the program above using Python's AST module.

Before continuing, take a moment to appreciate the fact that you have already learned a lot more than you knew prior to coming in contact with this reading. You are learning how to use Python's AST module. You may be noticing some patterns in the way that we build AST's. That's great! You are well on your way toward being confident and competent, in regards to this tough subject.

Writing Functions and Flow Control

Now that we're able to create constants, datatypes and assign functions; let's up the ante a little bit and learn to write functions. The exercise at the end of this part will be to write the `does_list_contain` function. Thus we will need to learn a few things:

1. How to write a function definition
2. How to use the logical operator `if`
3. How to write a `for` loop

Once we have these fundamentals, we can combine them to write the function:

```
def does_list_contain(item: str, list: list[str]) -> None:
    for list_item in list:
        if item == list_item:
            return True
    return False
```

`ast.FunctionDef`

When we need to represent a function using ASTs, we will use the `ast.FunctionDef` class. This class takes 7 parameters. Many more than the classes we've used until now. We, however, will only be focusing on 4 of these parameters, namely: `name`, `args`, `body`, and `returns`.

The `name` argument is where we define the name of the function that we are creating. This is simply a string value. For example, an AST defining the above function would have `name = "does_list_contain"`.

Next is the `args` argument. This is the most involved argument that the function definition has. This specifies all the things we intend to pass to the function. To specify the `args` we need to pass it `ast.arguments`. This `ast.arguments()` also has 7 parameters, but we will only use 2, `args` and `defaults`. Where `args` is a list of `ast.arg` elements. An `ast.arg` element has parameters: `arg` - the arguments name as a string, an `annotation` - which we will ignore, and a `type_comment` which is *optional*, but we will use.

That probably seemed like a lot, but read the paragraph back and identify each part in the below function and its `ast.FunctionDef`. This mini exercise will help the puzzle pieces fall into place.

```
def concatenate_strings(string_one, string_two) -> str:
        result_string = string_one + string_two
        return result_string
```

```
ast.FunctionDef(
        name="concatenate_strings",
        args=ast.arguments(
                args=[
                        ast.arg(arg="string_one", type_comment="str"),
```

```
                        ast.arg(arg="string_one", type_comment="str")
                ],
                defaults=[]
        ),
        body=[],
        decorator_list=[],
        returns="str"
)
```

The `body` of the `ast.arguments` node is just like the body of the `ast.Module()` class or any other `body` we come across. It is simply a place to *write code*.

The `decorator_list` argument takes a list of function decorators. We will not use this but it is required that we declare and define it. We do this by setting it equal to the empty list; `decorator_list=[]`.

Finally, we reach the last argument, `return`. This accepts a string description of the return type.

Now that we have all the pieces, we can write the function that we have above. In order to write the function above, we will also be doing an addition. Carefully observe each part of the addition operation, in its AST form, and attempt to understand it. Additionally, look for the return statement.

```
        ast.FunctionDef(
            name="concatenate_strings",
            args=ast.arguments(
                args=[
                    ast.arg(arg="string_one", type_comment="str"),
                    ast.arg(arg="string_two", type_comment="str")
                ],
                defaults=[]
            ),
            body=[
                ast.Assign(
                    targets=[ast.Name(id="result_string", ctx=ast.Store)],
                    value=ast.BinOp(
                        left=ast.Name(id='string_one', ctx=ast.Load()),
                        op=ast.Add(),
                        right=ast.Name(id='string_two', ctx=ast.Load())
                    )
                ),
                ast.Return(value=ast.Name(id="result_string", ctx=ast.Load()))
            ],
            decorator_list=[],
```

```
            returns="str"
        )
```

`ast.If`

There was a lot of information in the preceding subsection, so lets look at something smaller, the `ast.If` class. This class has three parts: `test` - which might hold a `ast.Compare` node, a body - with the code to execute if the condition is met, and `orelse` - with code to execute if the condition is not met.

> There are no `ast.Elif` nor `ast.Else` nodes in the AST module. If you want an `elif`, then add another `ast.If` inside of the `orelse` part. If you want an `else`, use the `orelse` part without an `ast.If` inside it.

```
number = 1
if(number == 1):
        return True
else:
        return False
```

```
ast.Module(
    body=[
        ast.Assign(
            targets=[ast.Name(id="number", ctx=ast.Store())],
            value=ast.Constant(value=1)
        ),
        # if-else statement
        ast.If(
            test=ast.Compare(
                left=ast.Name(id="number", ctx=ast.Load),
                ops=[ast.Eq()],
                comparators=[ast.Constant(1)]
            ),
            body=[
                ast.Return(value=ast.Constant(True))
            ],
            orelse=[
                ast.Return(value=ast.Constant(False))
            ]
        )
    ]
)
```

`ast.For`

Up until now, we have looked at every parameter and explained what we should assign to it. But now, we are starting to see the same nodes come up quite often: `ast.Name`, `ast.arg`, `ast.Constant`, etc.. From here forward we will not explain what these are and will not spend time going over each parameter and the corresponding AST node that you should assign it. Instead, we will be provided a skeleton from which we can deduce that information.

```
ast.For(
        target=ast.Name(),
        iter=ast.Name(),
        body=[],      # ASTs that make up the for loop body
        orelse=[],    # code to run after the loop (optional)
        type_comment= # string (optional)
)
```

Suppose that we have a collection of *numbers* in a list called `my_numbers` and that for some reason we want to add `1` to each of those numbers. We plan to accomplish this in a for loop. Follows is an example of how that for loop can be represented using the `ast.For` node and the code that it generates.

```
ast.For(
        target=ast.Name(id='number', ctx=ast.Store()),  # note: ast.Store()
        iter=ast.Name(id='my_numbers', ctx=ast.Load()),  # note: ast.Load()
        body=[
                ast.Assign(
                        targets=[ast.Name(id='number', ctx=ast.Load())],
                        value=ast.BinOp(
                        left=ast.Name(id='number', ctx=ast.Load()),
                        op=ast.Add(),
                        right=ast.Constant(1)
                        )
                )
        ],
        orelse=[
                ast.Call(
                        func=ast.Name(id="print"),
                        ctx=ast.Load(),
                        args=[ast.Name(id="my_list", ctx=ast.Load())],
                        keywords=[]
                )
        ],
```

```
        type_comment="int"
)
```

> It's important to note that the numbers that we are incrementing in the for loop are copies of the numbers in the list and *not* the actual numbers in the list. Thus manipulations we make to the elements inside the for loop will not be reflected in the original list. Therefore, the print statement after the loop will print the original list without the numbers incremented.

The code that is produced by this AST might look a little strange. In particular, the seemingly uncalled for `else` statement. But rest assured, the code will run as if that `else` statement weren't there at all. Here is what the preceding code produces:

```
for number in my_numbers:
    number = number + 1
else:print(my_list)
```

Exercise:

Using the tools we've learned, reproduce the `does_list_contain` function from the example program at the top.

```
def does_list_contain(item: str, list: list[str]) -> None:
        for list_item in list:
                if item == list_item:
                        return True
        return False
```

Building Classes from ASTs

Classes are used prolifically within the Python language. They are a fundamental feature of the programming language. It would be foolish not to learn how to represent them with AST's. Hence this section will be all about **building classes from ASTs**.

`ast.ClassDef`

Essential to building classes using the AST module is the `ast.ClassDef` node. This node is what defines the class entirely. The `ast.ClassDef` node takes 6 parameters, only one of which is new; `bases`. The parameter `bases` is a list of `ast.Name` nodes which explicitly enumerate the base classes of the current class. Given that we only define one class in the program above, we will not be using this parameter; we will assign `bases=[]` in our class definition. Follows is a skeleton of an `ast.ClassDef`:

```
ast.classDef(
        name="ClassName",  # string: name of the class we are defining
        bases=[],          # list ast.Name() nodes for each base class
        keywords=[],       # list of ast.keyword() nodes for any keywords
        body=[],           # ASTs that make up the class body
        decorator_list=[], # list of ast.Name() nodes for any decorators
        type_params=[]     # list of type parameters
)
```

Let's now define a `class Square` which takes one argument `side_length` to initialize, has a function called `update_side_length` which does what it says and returns nothing, and lastly a class function `area` which returns the area of the `Square` object.

Like much of what we have seen, the majority of the code will fall in the body parameter; almost all other arguments are auxiliary.

```
ast.ClassDef(
        name="Square",      # string: name of the class
        bases=[],           # no classes derive from Square
        keywords=[],        # no keywords are defined
        body=[],
        decorator_list=[],  # no class decorators are used
        type_params=[]
)
```

Until now we have been defining everything in one place. But earlier, we mentioned that we can build ASTs and assign them to variables. Then use those variables to *"copy and paste"* the AST into another AST. Because the class AST would be so huge on its own, lets write some ASTs for each of the the class functions separately, then place them into the `body` of the `ast.ClassDef`. We already know how to write an `ast.FunctionDef`, so the three class functions we need will be plopped down below with no further explanation.

> Check out how we refer to class member variables just like we would any other variable; the only difference being the prepended `self` moniker.

```
"""Note all class functions take self as their first argument"""

# The Square classes __init__ function
square_initializer = ast.FunctionDef(
        name="__init__",
        args=ast.arguments(
            args=[
                ast.arg(arg="self"),
```

```
                ast.arg(arg="side_length", type_comment="float")
            ],
            defaults=[]
        ),
        body=[
            ast.Assign(
                targets=[ast.Name(id="self.side_length", ctx=ast.Store())],
                value=ast.Name(id="side_length", ctx=ast.Load()),
                type_comment="float"
                )
        ],
        decorator_list=[],
        returns="None"
    )


# update the side length of the Square
update_side_length = ast.FunctionDef(
        name="update_side_length",
        args=ast.arguments(
            args=[
                ast.arg(arg="self"),
                ast.arg(arg="side_length", type_comment="float")
            ],
            defaults=[]
        ),
        body=[
            ast.Assign(
                targets=[ast.Name(id="self.side_length", ctx=ast.Store())],
                value=ast.Name(id="side_length", ctx=ast.Load()),
                type_comment="float"
                )
        ],
        decorator_list=[],
        returns="None"
    )


# get the are of a Square
area = ast.FunctionDef(
    name='area',
    args=ast.arguments(
        args=[
            ast.arg(arg='self'),
        ],
        defaults=[]  # no default values
```

```
    ),
    body=[
        ast.Return(
            value=ast.BinOp(
                left=ast.Name(id='self.side_length', ctx=ast.Load()),
                op=ast.Mult(),
                right=ast.Name(id='self.side_length', ctx=ast.Load())
            )
        )
    ],
    decorator_list=[],
    returns='float'
)
```

Now that we have the contents of the body of our square class saved in the variables `square_initializer`, `update_side_length`, and `area`; respectively. We can easily plug these values into our `ast.ClassDef` skeleton. The result will be the class definition for our toy class.

```
ast.ClassDef(
    name="Square",  # string: name of the class we are defining
    bases=[],       # no base classes
    keywords=[],    # list of ast.keyword() nodes for any keywords
    body=[
            square_initializer,  # Square's __init__ function
            update_side_length,  # Square's update_side_length function
            area                 # Square's area function
        ],
    decorator_list=[],  # no decorators
    type_params=[]      # Not applicable
)
```

Final Exercise:

Recreate the file at the begining of this reading.

Congratulations!!! At this point you have all the necessary skills to represent the above program using the AST module from python. It is a great idea to have the *Abstract Syntax Trees* documentation at your side while you build your first AST's for whatever project you intend to do. The AST module's documentation will read much easier now that you have the strong fundamentals required to use ASTs with competence and confidence. Happy Coding!