PLY & Conda
To implement compiler

Prepared by

Zeyad moneer abo El-Zahab

Agenda

- ❑ What is Anaconda
- ❑ Using Conda
- ❑ What is RPLY?
- ❑ What is LLVMlite?
- ❑ Integrating RPLY and LLVMlite
- ❑ EBNF vs. BNF
- ❑ How to write code
  - ✓ Lexer
  - ✓ Parser
  - ✓ Code Generator

What is Anaconda?

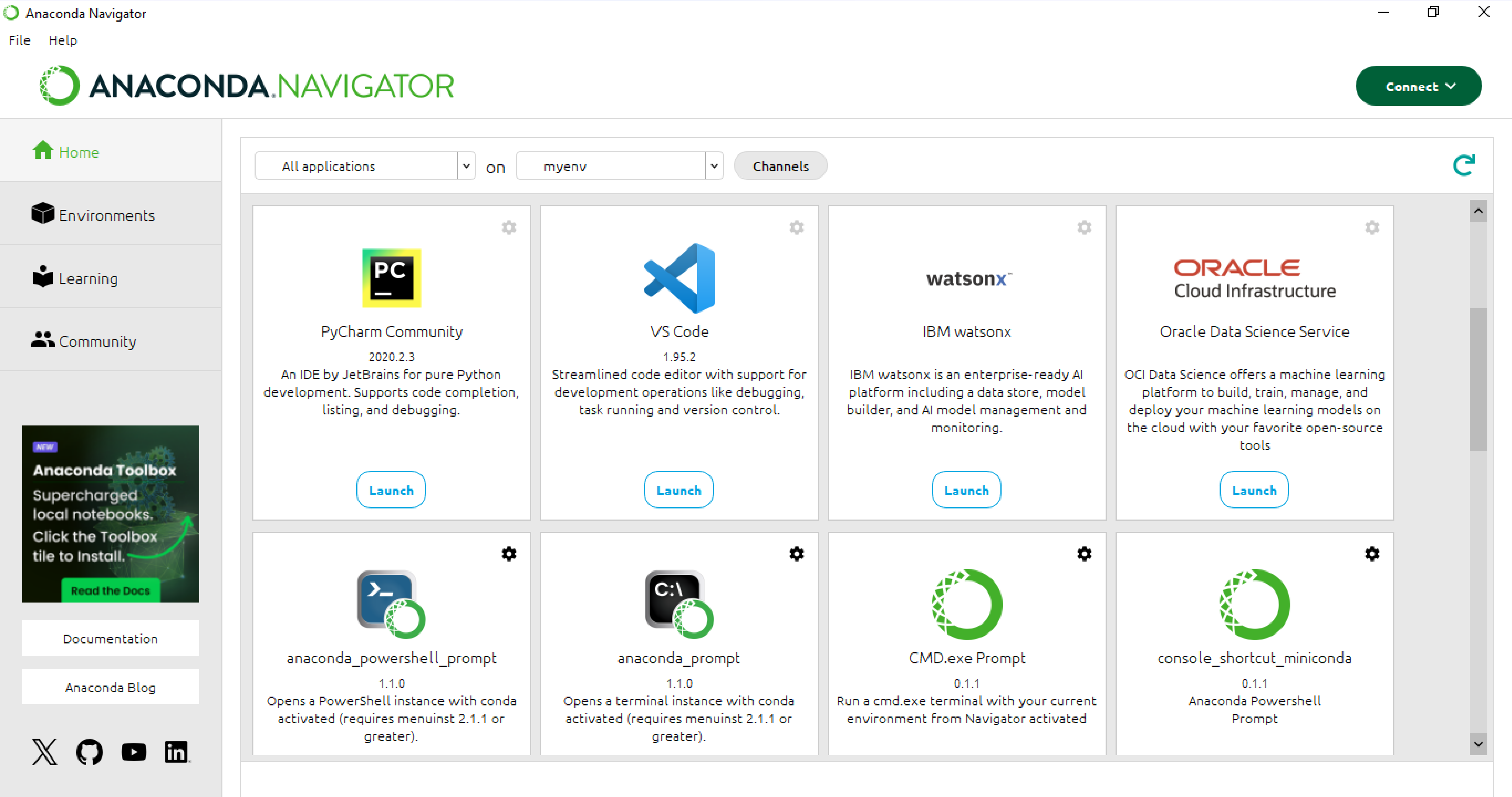
- Anaconda is a distribution of the Python and R programming languages for scientific computing and data science.
- It includes hundreds of popular data science packages and tools, making it an all-in-one solution for data analysis, machine learning, and scientific computing.
- Anaconda simplifies the installation and management of these packages, allowing data scientists to focus on their work rather than spending time on configuration and setup.

Downloading and Installing Anaconda

- Step 1:
  Visit the Anaconda website **(https://www.anaconda.com/)** and download the appropriate installer for your operating system (Windows, macOS, or Linux).

- Step 2:
  Run the installer and follow the on-screen instructions to complete the installation process. Anaconda will automatically set up your Python environment and install a wide range of data science packages.

- Step 3:
  Once installed, you can launch the Anaconda Navigator, a graphical user interface that provides access to Jupyter Notebook, Spyder, and other Anaconda tools and applications.



Anaconda Navigator
File
Help
ANACONDA.NAVIGATOR
Connect
Home
Environments
Learning
Community
All applications
on
myenv
Channels
PyCharm Community
2020.2.3
An IDE by JetBrains for pure Python development. Supports code completion, listing, and debugging.
Launch
VS Code
1.95.2
Streamlined code editor with support for development operations like debugging, task running and version control.
Launch
watsonx
IBM watsonx
IBM watsonx is an enterprise-ready AI platform including a data store, model builder, and AI model management and monitoring.
Launch
ORACLE
Cloud Infrastructure
Oracle Data Science Service
OCI Data Science offers a machine learning platform to build, train, manage, and deploy your machine learning models on the cloud with your favorite open-source tools
Launch
NEW
Anaconda Toolbox
Supercharged local notebooks.
Click the Toolbox tile to install.
Read the Docs
Documentation
Anaconda Blog
anaconda_powershell_prompt
1.1.0
Opens a PowerShell instance with conda activated (requires menuinst 2.1.1 or greater).
C:\
anaconda_prompt
1.1.0
Opens a terminal instance with conda activated (requires menuinst 2.1.1 or greater).
CMD.exe Prompt
0.1.1
Run a cmd.exe terminal with your current environment from Navigator activated
console_shortcut_miniconda
0.1.1
Anaconda Powershell Prompt

Using Conda

- Once installed, you can use Conda from your terminal or command prompt. Here are some basic commands:

  - ✓ **Creating a New Environment**

    ```
    conda create -n my_env python=3.9
    ```

  - ✓ **Activating an Environment**

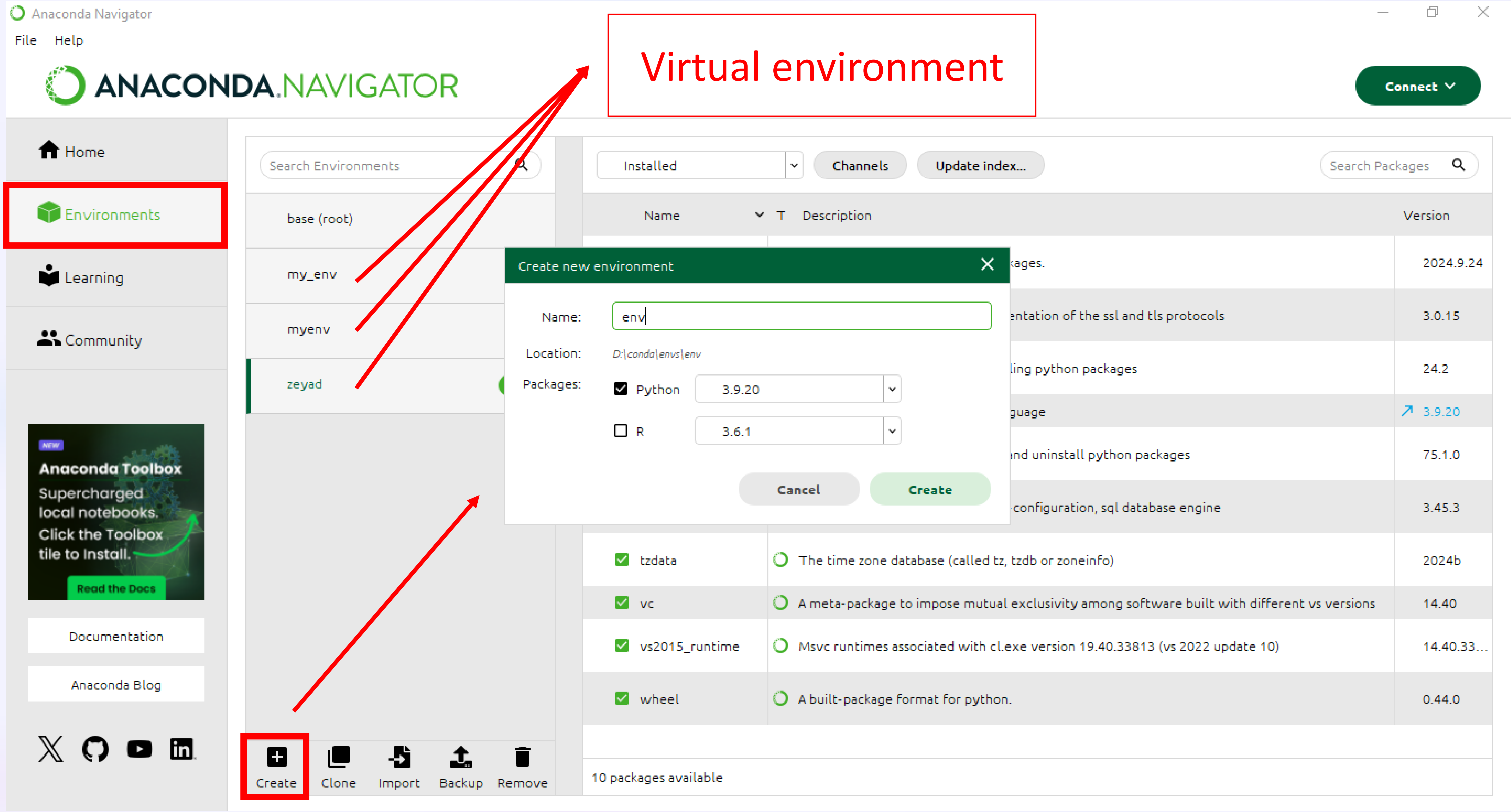
    ```
    conda activate my_env
    ```

  - ✓ **Deactivating an Environment**

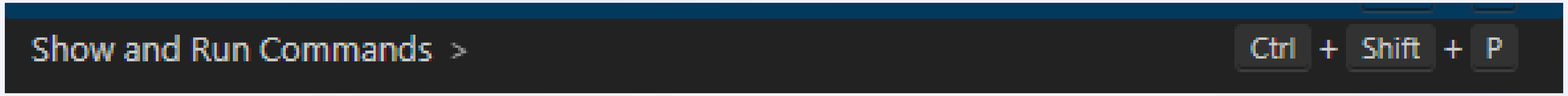
    ```
    conda deactivate
    ```

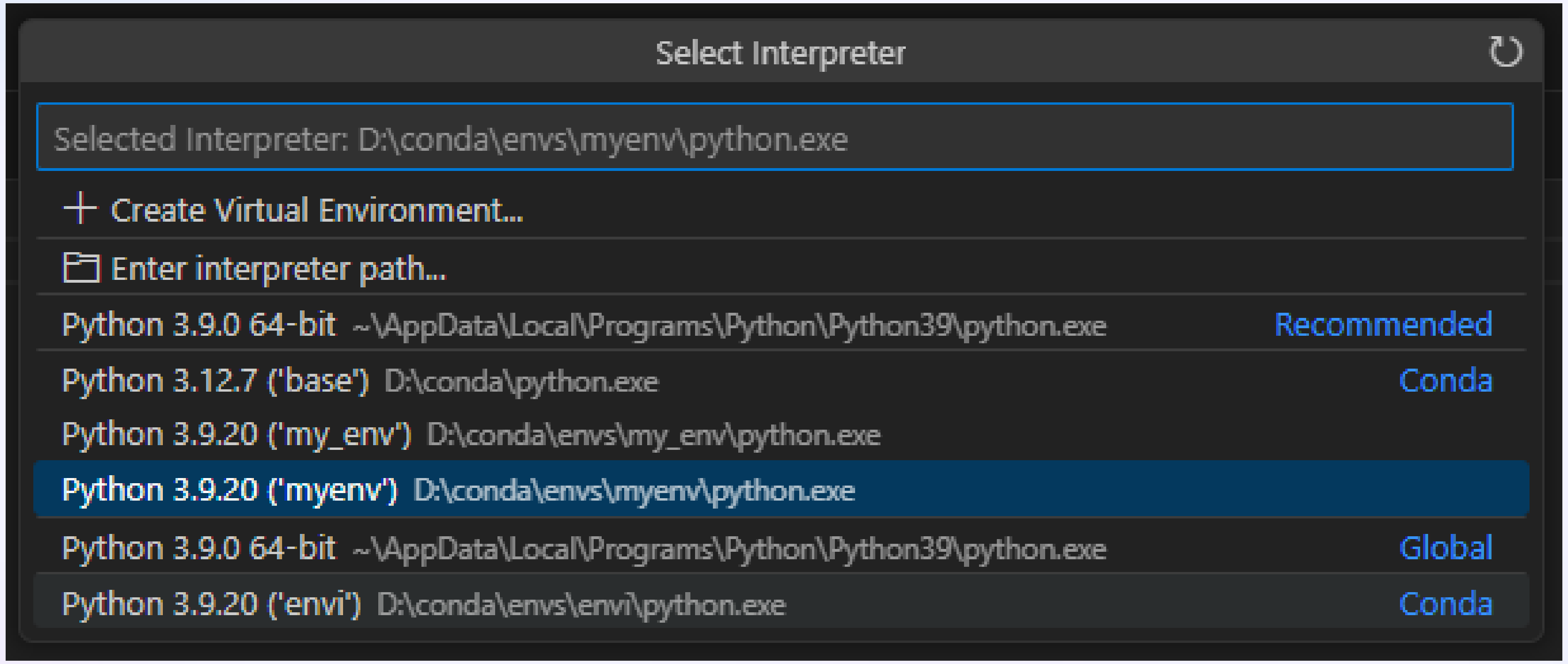
  - ✓ **Installing Packages**

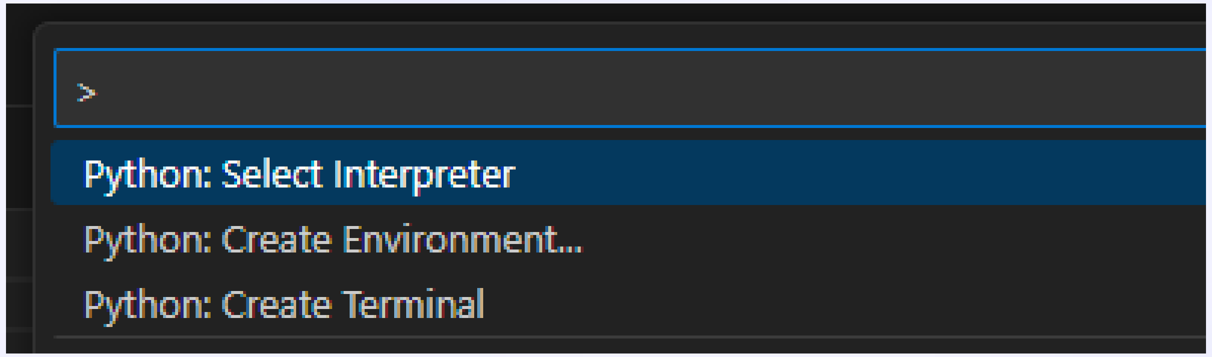
    ```
    conda install package_name
    ```


Anaconda Navigator
File
Help
ANACONDA.NAVIGATOR
Virtual environment
Connect
Home
Environments
Learning
Community
Anaconda Toolbox
Supercharged local notebooks.
Click the Toolbox tile to install.
Read the Docs
Documentation
Anaconda Blog
Search Environments
base (root)
my_env
myenv
zeyad
Create
Clone
Import
Backup
Remove
Installed
Channels
Update index...
Search Packages
Name
T
Description
Version
Create new environment
Name:
env
Location:
D:\conda\envs\env
Packages:
Python
3.9.20
R
3.6.1
Cancel
Create
2024.9.24
3.0.15
24.2
3.9.20
75.1.0
3.45.3
tzdata
The time zone database (called tz, tzdb or zoneinfo)
2024b
vc
A meta-package to impose mutual exclusivity among software built with different vs versions
14.40
vs2015_runtime
Msvc runtimes associated with cl.exe version 19.40.33813 (vs 2022 update 10)
14.40.33...
wheel
A built-package format for python.
0.44.0
10 packages available

Python (Virtual environment) in VS Code







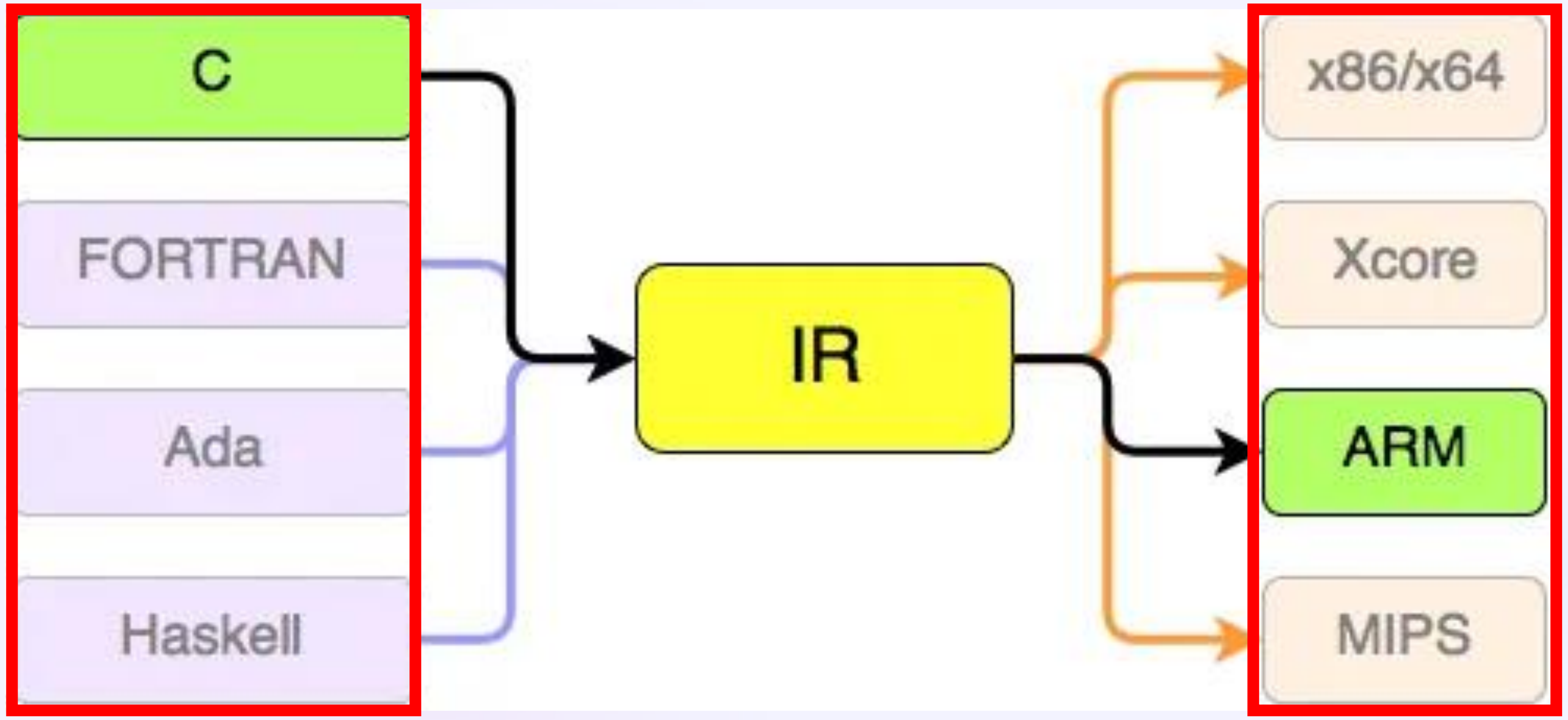
What is RPLY?

- RPLY (Recursive Descent Parser Library for Python) is a powerful Python library that allows you to create custom parsers. It provides a simple and intuitive interface to define grammars and implement parsing logic.
- Similar in functionality to PLY but simpler and more lightweight.
- Used for creating interpreters and compilers

What is LLVMlite?

- LLVMlite is a Python library that provides a high-level interface to the LLVM Intermediate Representation (IR). It allows you to create and manipulate LLVM IR, which is a low-level representation of code that can be optimized and translated into machine code.

Source Language



Target Architectures

Integrating RPLY and LLVMlite

- Use RPLY for parsing high-level language constructs.
- Translate parsed constructs into LLVM IR (Intermediate Representation).
- Leverage LLVMlite for optimization and code generation.
- Applications: custom interpreters, compilers, and dynamic analysis tools.

EBNF vs. BNF

EBNF (Extended Backus-Naur Form)

EBNF is an extension of the Backus-Naur Form (BNF) that provides a more expressive and flexible way to define the syntax of programming languages. It includes additional operators and constructs, such as optional and repeated elements, which make it easier to describe complex grammar rules.

BNF (Backus-Naur Form)

BNF is a formal notation used to describe the syntax of programming languages. It uses a set of production rules to define the structure of a language, making it a powerful tool for language designers and parser generators.

EBNF vs. BNF

Notation	Usage
Definition	=
Concatenation	,
termination	;
optional	[...]
repetition	{ ... }
terminal string	" ... "
special sequence	? ... ?
exception	–
comment	(* ... *)

How to Write code

How To write Code

- Our compiler can be divided into three components:
  - ✓ Lexer
  - ✓ Parser
  - ✓ Code Generator
- For the Lexer and Parser we’ll be using RPLY, really similar to PLY but with a better API. And for the Code Generator, we’ll use LLVMlite, a Python library for binding LLVM components.

LEXER

- let’s start coding our compiler.

  First, create a file named lexer.py.

  We’ll define our tokens on this file.

  We’ll only use LexerGenerator class

  from RPLY to create our Lexer.

```python
from rply import LexerGenerator

class Lexer():
    def __init__(self):
        self.lexer = LexerGenerator()

    def _add_tokens(self):
        # Print
        self.lexer.add('PRINT', r'print')
        # Parenthesis
        self.lexer.add('OPEN_PAREN', r'\(')
        self.lexer.add('CLOSE_PAREN', r'\)')
        # Semi Colon
        self.lexer.add('SEMI_COLON', r'\;')
        # Operators
        self.lexer.add('SUM', r'\+')
        self.lexer.add('SUB', r'\-')
        # Number
        self.lexer.add('NUMBER', r'\d+')
        # Ignore spaces
        self.lexer.ignore('\s+')

    def get_lexer(self):
        self._add_tokens()
        return self.lexer.build()
```

AST File

- To implement our parser, we’ll use the structure created with out EBNF as model. Luckly, RPLY’s parser uses a format really similar to the EBNF to create it’s parser.
- First, create a new file named ast.py . It will contain all classes that are going to be called on the parser and create the AST.
- Second, we need to create the parser. For that, we’ll use ParserGenerator from RPLY. Create a file name parser.py:

```python
class Number():
    def __init__(self, value):
        self.value = value

    def eval(self):
        return int(self.value)


class BinaryOp():
    def __init__(self, left, right):
        self.left = left
        self.right = right


class Sum(BinaryOp):
    def eval(self):
        return self.left.eval() + self.right.eval()


class Sub(BinaryOp):
    def eval(self):
        return self.left.eval() - self.right.eval()


class Print():
    def __init__(self, value):
        self.value = value

    def eval(self):
        print(self.value.eval())
```

AST File

```python
from rply import ParserGenerator
from ast_1 import Number, Sum, Sub, Print
from lexer import Lexer

class Parser():
    def __init__(self):
        self.pg = ParserGenerator(
            # A list of all token names accepted by the parser.
            ['NUMBER', 'PRINT', 'OPEN_PAREN', 'CLOSE_PAREN',
             'SEMI_COLON', 'SUM', 'SUB']
        )

    def parse(self):
        @self.pg.production('program : PRINT OPEN_PAREN expression CLOSE_PAREN SEMI_COLON')
        def program(p):
            return Print(p[2])

        @self.pg.production('expression : expression SUM expression')
        @self.pg.production('expression : expression SUB expression')
        def expression(p):
            left = p[0]
            right = p[2]
            operator = p[1]
            if operator.gettokentype() == 'SUM':
                return Sum(left, right)
            elif operator.gettokentype() == 'SUB':
                return Sub(left, right)

        @self.pg.production('expression : NUMBER')
        def number(p):
            return Number(p[0].value)

        @self.pg.error
        def error_handle(token):
            raise ValueError(token)

    def get_parser(self):
        return self.pg.build()
```

Parser File

Parser (cont.)

- Now, you'll see the output being the result of print(4 + 4 - 2), which is equal to printing 6.

```
#-----------------------------------
# our file using lexer and parser

text_input = """
print(4 + 4 - 2);
"""

lexer = Lexer().get_lexer()
tokens = lexer.lex(text_input)

pg = Parser()
pg.parse()
parser = pg.get_parser()
parser.parse(tokens).eval()
```

Code Generator

- The third and last component of out compiler is the Code Generator. It’s role is to transform the AST created from the parser into machine language or an IR. In this case, it’s going to transform the AST into LLVM IR.
- There aren’t good guides on how to implement code generation with LLVM on Python.
- LLVMlite doesn’t have a implementation to a print function, so you have to define your own.

1

```python
from llvmlite import ir, binding

class CodeGen():
    def __init__(self):
        self.binding = binding
        self.binding.initialize()
        self.binding.initialize_native_target()
        self.binding.initialize_native_asmprinter()
        self._config_llvm()
        self._create_execution_engine()
        self._declare_print_function()

    def _config_llvm(self):
        # Config LLVM
        self.module = ir.Module(name=__file__)
        self.module.triple = self.binding.get_default_triple()
        func_type = ir.FunctionType(ir.VoidType(), [], False)
        base_func = ir.Function(self.module, func_type, name="main")
        block = base_func.append_basic_block(name="entry")
        self.builder = ir.IRBuilder(block)

    def _create_execution_engine(self):
        """
        Create an ExecutionEngine suitable for JIT code generation on
        the host CPU.  The engine is reusable for an arbitrary number of
        modules.
        """
        target = self.binding.Target.from_default_triple()
        target_machine = target.create_target_machine()
        # And an execution engine with an empty backing module
        backing_mod = binding.parse_assembly("")
        engine = binding.create_mcjit_compiler(backing_mod, target_machine)
        self.engine = engine
```

2

```python
def _declare_print_function(self):
    # Declare Printf function
    voidptr_ty = ir.IntType(8).as_pointer()
    printf_ty = ir.FunctionType(ir.IntType(32), [voidptr_ty], var_arg=True)
    printf = ir.Function(self.module, printf_ty, name="printf")
    self.printf = printf

def _compile_ir(self):
    """
    Compile the LLVM IR string with the given engine.
    The compiled module object is returned.
    """
    # Create a LLVM module object from the IR
    self.builder.ret_void()
    llvm_ir = str(self.module)
    mod = self.binding.parse_assembly(llvm_ir)
    mod.verify()
    # Now add the module and make sure it is ready for execution
    self.engine.add_module(mod)
    self.engine.finalize_object()
    self.engine.run_static_constructors()
    return mod

def create_ir(self):
    self._compile_ir()

def save_ir(self, filename):
    with open(filename, 'w') as output_file:
        output_file.write(str(self.module))
```

Code Generator (cont.)

- let’s update our main.py file to call CodeGen methods:
- I removed the input program from this file and created a new file called input.toy to simulate a external program. It’s content is the same as the input described.

```python
from lexer import Lexer
from parser import Parser
from codegen import CodeGen

fname = "input.toy"
with open(fname) as f:
    text_input = f.read()

lexer = Lexer().get_lexer()
tokens = lexer.lex(text_input)

codegen = CodeGen()

module = codegen.module
builder = codegen.builder
printf = codegen.printf

pg = Parser(module, builder, printf)
pg.parse()
parser = pg.get_parser()
parser.parse(tokens).eval()

codegen.create_ir()
codegen.save_ir("output.ll")
```

Code Generator (cont.)

- Another change that was made is passing module, builder and printf objects to the Parser. This was made so we could pass this objects to the AST, where the LLVM AST is created. So, we change parser.py to receive these objects and pass them to the AST.

NEW Parser File

```python
from rply import ParserGenerator
from ast import Number, Sum, Sub, Print


class Parser():
    def __init__(self, module, builder, printf):
        self.pg = ParserGenerator(
            # A list of all token names accepted by the parser.
            ['NUMBER', 'PRINT', 'OPEN_PAREN', 'CLOSE_PAREN',
             'SEMI_COLON', 'SUM', 'SUB']
        )
        self.module = module
        self.builder = builder
        self.printf = printf

    def parse(self):
        @self.pg.production('program : PRINT OPEN_PAREN expression CLOSE_PAREN SEMI_COLON')
        def program(p):
            return Print(self.builder, self.module, self.printf, p[2])

        @self.pg.production('expression : expression SUM expression')
        @self.pg.production('expression : expression SUB expression')
        def expression(p):
            left = p[0]
            right = p[2]
            operator = p[1]
            if operator.gettokentype() == 'SUM':
                return Sum(self.builder, self.module, left, right)
            elif operator.gettokentype() == 'SUB':
                return Sub(self.builder, self.module, left, right)

        @self.pg.production('expression : NUMBER')
        def number(p):
            return Number(self.builder, self.module, p[0].value)

        @self.pg.error
        def error_handle(token):
            raise ValueError(token)

    def get_parser(self):
        return self.pg.build()
```

NEW AST File

```python
class Number():
    def __init__(self, value):
        self.value = value

    def eval(self):
        return int(self.value)


class BinaryOp():
    def __init__(self, left, right):
        self.left = left
        self.right = right


class Sum(BinaryOp):
    def eval(self):
        return self.left.eval() + self.right.eval()


class Sub(BinaryOp):
    def eval(self):
        return self.left.eval() - self.right.eval()


class Print():
    def __init__(self, value):
        self.value = value

    def eval(self):
        print(self.value.eval())
```

Thank you

Any questions?