Intermediate Python

Execution models

Execution models

- How is my code executed?
- Learning to think about code like Python does
- Code and data
- Values and types
- Syntax of a language
- Statements and expressions
- Operator precedence

Execution models?

"Execution model" refers to the behaviour of a language.

Execution models of a program

In order to understand what a program does, we must understand the execution model of the language we're working with.

Execution models help us answer questions

- What is *this* line of code doing?
- What is my code doing?
- What is my program doing?

Code and data

Code and data

When we write programs, we do so by writing code.

Writing code

We write `if` statements, `for` loops, variables, integers, strings, and so.

Our code are the instructions for Python

Print this.

Go there.

Do that.

Always do this.

Sometimes do that.

Code and data

When we run our programs, Python will execute the code we wrote.

Code and data

When we run our programs, Python will execute the instructions we provided.

Code and data

And as part of executing our code, it will create data.

Data

Data refers to the *values* in our program.

The *strings*, *integers*, *booleans*,

dictionaries, *lists*,

and even *functions*.

Code and data

Not all code is data, but all data can be represented as code.

Not all code is data

For example, `if` statements and `for` loops are just code.

But all data can be represented as code

- Strings: `"This is a string"`
- Integers: `123`
- Booleans: `True`
- Lists: `[7, 7, 7]`
- And so on.

Code and data

The code that we write is stored in files in our hard-drives and can be made to run multiple times.

Code and data

The data in our programs is stored in RAM and only "lives"
for as long as our program is running.

Values and types

All values have a "type"

A "type" refers to the kind of data the value is.

Types

- String
- Boolean
- Integer
- Float
- List
- Dictionary
- Function
- And more.

Values and types

All values have a "type".

Values and types, and code and data

Values *are* the data in our programs, therefore all data has a type.

Values and types, and code and data

Code that is not data does not have a type.

The type of a value

The type of a value helps us answer questions like,

"What is this?"

"What can I do with this?"

Source of a value

Every value must come from somewhere.

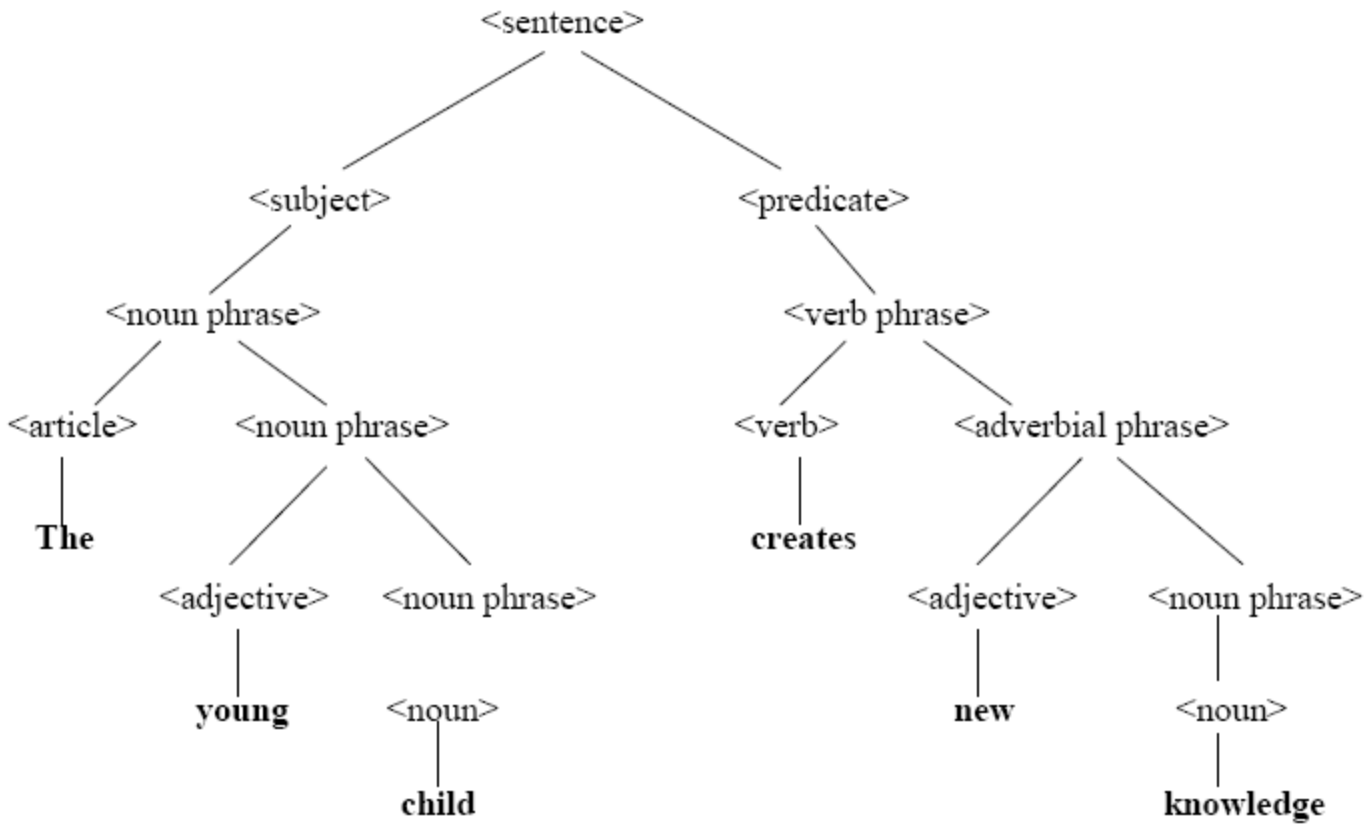
Every value must come from somewhere

Values must be declared, imported, or provided by Python. Otherwise you'll get this error `NameError: name '' is not defined`

Syntax of a language

Syntax of a language

The syntax of language refers to how the language can be written.



https://www.researchgate.net/figure/A-tree-representation-for-the-syntax-of-the-sentence-The-young-child-creates-new_fig2_320630490

How the language can be written

Knowing how a language can be written is important because
it will allow us to write code that is *correct*.

Code that is correct

Code is *correct* when it can be executed by Python, *and* when it does what we needed it to do.

Code that is correct

Writing code that is correct is not easy, but it is possible.

Code that is correct

Understanding the syntax of a language is the first step to understanding what a line of code will do, and therefore the first step to writing code that is correct.

Syntax of a language

A language's syntax is a set of rules that describe how code must be written. These rules are the language's grammar.

Specific example: by and until in English

1. Use *"by"* to talk about a deadline. When we use "by" we are only concerned about the deadline or the time when something or some period ends.
    - We need to submit this *by* 5 pm.
    - He must finish this *by* next week.
2. We use *"until"* to talk about the period of time from a starting point until a deadline or the end of some period of time. So, "by" only refers to the deadline, but "until" refers to all the time between now and the end of something.
    - I have to work *until* 10 pm.
    - The store is open *until* 1 am.

https://www.talkenglish.com/speaking/lessondetails.aspx?ALID=4398#:~:text=We use "until" to talk,report any day until Friday.

Generic example: verb/subject agreement in English

1. The verb in an or, either/or, or neither/nor sentence agrees with the noun or pronoun closest to it.
    - Neither the plates nor the serving bowl *goes* on that shelf.
    - Neither the serving bowl nor the plates *go* on that shelf.

https://www.grammarbook.com/grammar/subjectVerbAgree.asp

Punctuation example: colons in English

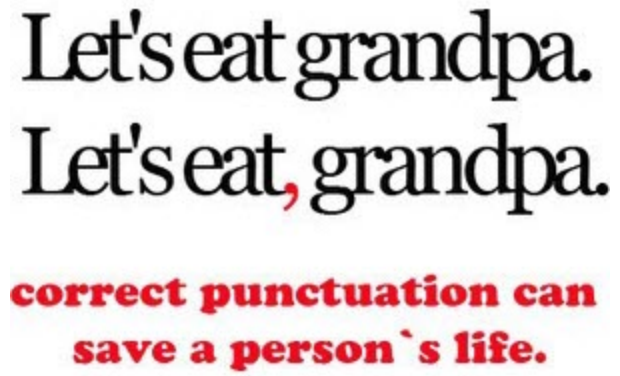
1. A colon can be used to separate two independent clauses when the second clause is directly related to the first clause (not just vaguely related), or when the emphasis is on the second clause.
    - There are three types of muscle in the body: cardiac, smooth, and skeletal.
2. Colons can be used before a noun or a noun phrase.
    - The movie had everything I wanted: action and suspense.

https://www.grammarly.com/blog/colon-2/

Grammar rules have impact on the meaning of a sentence

https://cybertext.files.wordpress.com/2011/03/comma.png

Grammar rules have impact on the meaning of a sentence



https://cybertext.files.wordpress.com/2011/03/comma.jpg

Programming languages also have a grammar

https://docs.python.org/3/reference/expressions.html

https://docs.python.org/3/reference/simple_stmts.html

https://docs.python.org/3/reference/compound_stmts.html

A language's grammar

A grammar is described in specific format. This format is called EBNG, or just BNF sometimes.

assignment statement

```
assignment_stmt ::=  (target_list "=")+ (starred_expression | yield_expression)
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                     | "(" [target_list] ")"
                     | "[" [target_list] "]"
                     | attributeref
                     | subscription
                     | slicing
                     | "*" target

identifier   ::=  xid_start xid_continue*
id_start     ::=  <all characters in ...>
id_continue  ::=  <all characters in ...>
xid_start    ::=  <all characters in ...>
xid_continue ::=  <all characters in ...>
```

https://docs.python.org/3/reference/simple_stmts.html

if statement

```
if_stmt ::=  "if" assignment_expression ":" suite
             ("elif" assignment_expression ":" suite)*
             ["else" ":" suite]
```

while loop

```
while_stmt ::=  "while" assignment_expression ":" suite
                ["else" ":" suite]
```

for loop

```
for_stmt ::=  "for" target_list "in" starred_list ":" suite
              ["else" ":" suite]
```

https://docs.python.org/3/reference/compound_stmts.html

Statements and expressions

Statements and expressions

Python's syntax is split into two types of structures: statements and expressions

Statements and expressions

What's the difference?

Statements and expressions

Expressions are values.

Expressions are values

If it can go in the right-hand side of an assignment, it's an expression.

Statements and expressions

Statements are everything else.

Let's use something that's a little simpler than EBNF

assignment statement

```
assignment_statement ::= <variable_name> "=" <expression>
```

if statement

```
if_statement ::= "if" <boolean_like_expression>
                 ":" <indented_code>
```

while loop

```
while_loop_statement ::= "while" <boolean_like_expression>
                         ":" <indented_code>
```

for loop

```
for_loop_statement ::= "for" <variable_name> "in" <list_like_expression>
                       ":" <indented_code>
```

expressions, literals, and variables

```
lit_expression ::= <literal_value>
                 | <expression> <arithmetic_operator> <expression>
                 | <expression> <boolean_operator> <expression>
                 | <not_operator> <expression>

comp_expression ::= <variable>
                  | <list_index_access>
                  | <attribute_reference>
                  | <function_call>

expression ::= <lit_expression>>
             | <comp_expression>

literal_value ::= <integer>
                | <float>
                | <dictionary>
                | <list>

variable ::= <any valid variable name>
```

dictionary, list, accessors, and function calls

```
dictionary ::= "{"
                   <string> ":" <expression> ","
                   <string> ":" <expression> ","
                   <string> ":" <expression> "," ...
               "}"

list ::= "["
             <expression> ","
             <expression> ","
             <expression> "," ...
         "]"

list_index_access ::= <list> "[" <integer> "]"
                    | <variable> "[" <integer> "]"

attribute_reference ::= <comp_expression> "." <variable>
                      | <comp_expression> "." <function_call>

function_call ::= <comp_expression> "(" <expression> "," <expression> "," ... ")"
```

Partial operator precedence

Order	Operator	Description
1	`(), [], {}`	Parenthesized expression, list, dictionary
2	`x[index], x(arguments...), x.attribute`	Subscription, call, attribute reference
3	`**`	Exponentiation
4	`*, @, /, //, %`	Multiplication, matrix multiplication, division, floor division, remainder
5	`+, -`	Addition and subtraction

https://docs.python.org/3/reference/expressions.html

Partial operator precedence (cont.)

Order	Operator	Description
6	`in, not in, is, is not, <, <=, >, >=, !=, ==`	Comparisons, including membership tests and identity tests
7	`not x`	Boolean NOT
8	`and`	Boolean AND
9	`or`	Boolean OR

https://docs.python.org/3/reference/expressions.html