Midterm 1: A Brief Overview of Python

by Group 8

Tong Zhao, Xiaowei Yang, Yun-Chien Wu, Stephen Trotter, Jonathan Toombs,
Mathew Sedgewick, Austin Hendy, Adar Guy, Marin Evergreen

Q1: Mathew Sedgwick, Xiaowei Yang
Q2: Jonathan Toombs, Austin Hendy
Q3: Adar Guy, Marin Evergreen
Q4:Tong Zhao, Yun-Chien Wu
Editor: Stephen Trotter

1. History

Brief History of Python (Mathew Sedgwick)

Python was first conceived by CWI's Guido van Rossum in December of 1989[1]. The inspiration for this language was a frustration, felt by van Rossum, towards the ABC programming language he was working with. This was exacerbated by the fact that ABC could not be extended to remedy these frustrations [1]. Shortly after a full year of development, van Rossum released his Python 0.9.0 to alt.sources [2], an online source-code repository. Python 2.0 was released in October of 2000 with new features, including a cycle-detecting garbage collector and support for Unicode [3]. Today, the most recent Python programming language now comes with an extensive standard library that includes the capabilities for string processing, Internet protocols, software engineering, and operating system interfaces [4].

Syntax and Semantics (Mathew Sedgwick)

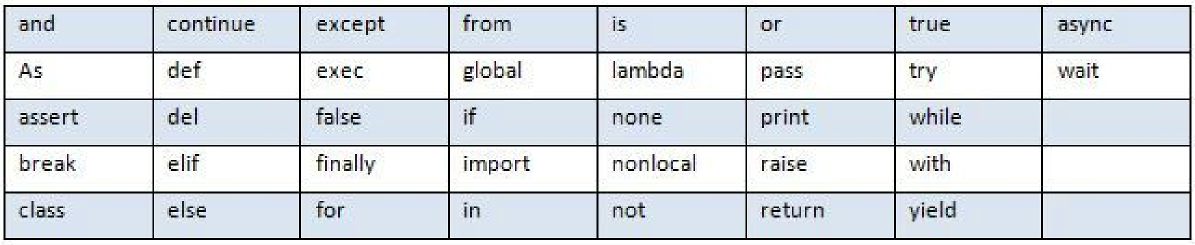
Python was designed to have both a simple and consistent syntax; because of this, it has been suggested that it may be better than Pascal, C, C++, and Java as a beginner's programming language [5]. Van Rossum wrote that "Modula-3 is the origin of the syntax and semantics used for exceptions, and some other Python features" [1]. This is evident through comparison of Python and Modula-3 syntax and semantics. For example, by looking at Table 1 [6] and Table 2[7], one can see the similar, simplistic language used for the keywords in Python and Modula-3; these tables can be further compared with the C programming language's keywords (Table 3)[8], where it is apparent that C is a lower-leveled language with respect to syntax. Python also inherits the whitespace indentation method from ABC to delimit its program blocks. This method of program block delimiting is very much unlike the "free-format" style that was taken on by many other modern programming languages, including C and Java. Examples of code block delimiting by different languages can be shown in Figure 1. Python's semantics, however, took more after those of C; while still using basic arithmetic operators like Modula-3, it switched from Modula-3's assignment operator ':=' to what C uses, and what is most commonly found today in modern programming languages, '='. Similarly, Python switched to C's '==' comparison operator from Modula-3's '='.

Pragmatics (Mathew Sedgwick)

In programming, pragmatics refers to "practical aspects of how constructs and features of a language may be used to achieve various objectives" [11]. In simple, high-level terms, pragmatics refers to the practicality of the language and how it can be used to accomplish tasks in simple, powerful ways. "The Zen of Python" (Figure

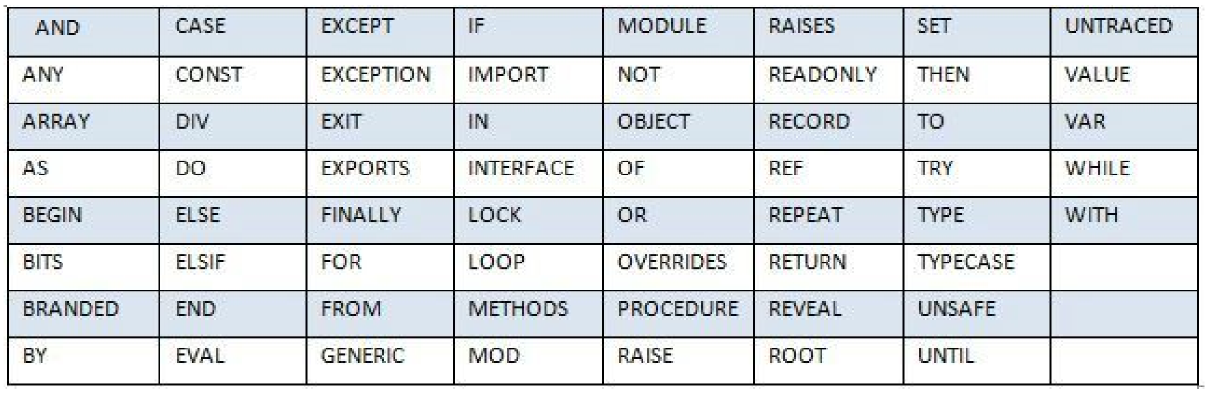
2)[12] reads aphorisms that depict the core philosophy of Python and represent its' pragmatism. In particular, line thirteen that reads "there should be one-- and preferably only one --obvious way to do it," blatantly opposes Perl's approach: "there is more than one way to do it" [13].

Table 1. Python's keywords.



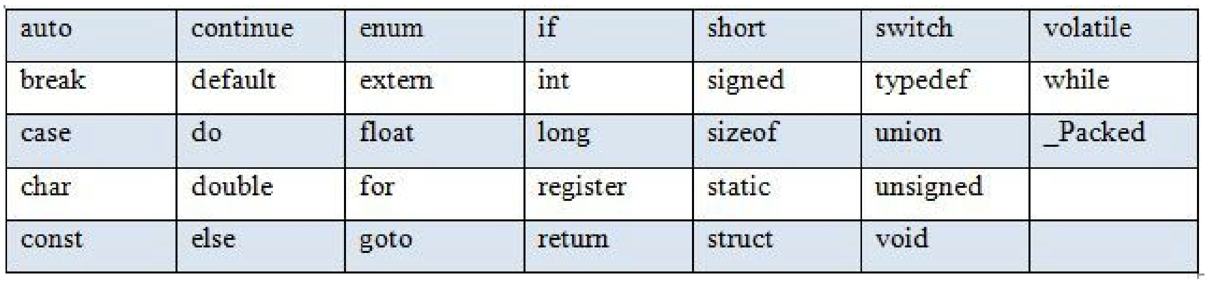
and	continue	except	from	is	or	true	async
As	def	exec	global	lambda	pass	try	wait
assert	del	false	if	none	print	while	
break	elif	finally	import	nonlocal	raise	with	
class	else	for	in	not	return	yield	

Table 2. Modula-3's keywords.



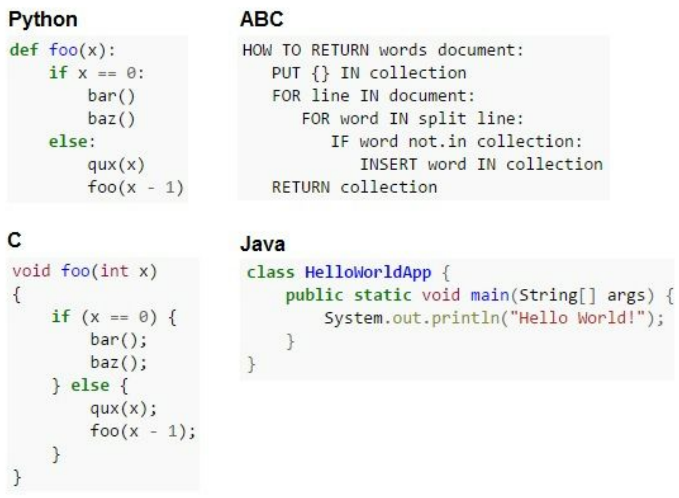
AND	CASE	EXCEPT	IF	MODULE	RAISES	SET	UNTRACED
ANY	CONST	EXCEPTION	IMPORT	NOT	READONLY	THEN	VALUE
ARRAY	DIV	EXIT	IN	OBJECT	RECORD	TO	VAR
AS	DO	EXPORTS	INTERFACE	OF	REF	TRY	WHILE
BEGIN	ELSE	FINALLY	LOCK	OR	REPEAT	TYPE	WITH
BITS	ELSIF	FOR	LOOP	OVERRIDES	RETURN	TYPECASE	
BRANDED	END	FROM	METHODS	PROCEDURE	REVEAL	UNSAFE	
BY	EVAL	GENERIC	MOD	RAISE	ROOT	UNTIL	

Table 3. C's keywords.



auto	continue	enum	if	short	switch	volatile
break	default	extern	int	signed	typedef	while
case	do	float	long	sizeof	union	_Packed
char	double	for	register	static	unsigned	
const	else	goto	return	struct	void	

Figure 1. Code delimiting examples by language.



Python

```python
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x - 1)
```

ABC

```
HOW TO RETURN words document:
   PUT {} IN collection
   FOR line IN document:
      FOR word IN split line:
         IF word not.in collection:
            INSERT word IN collection
   RETURN collection
```

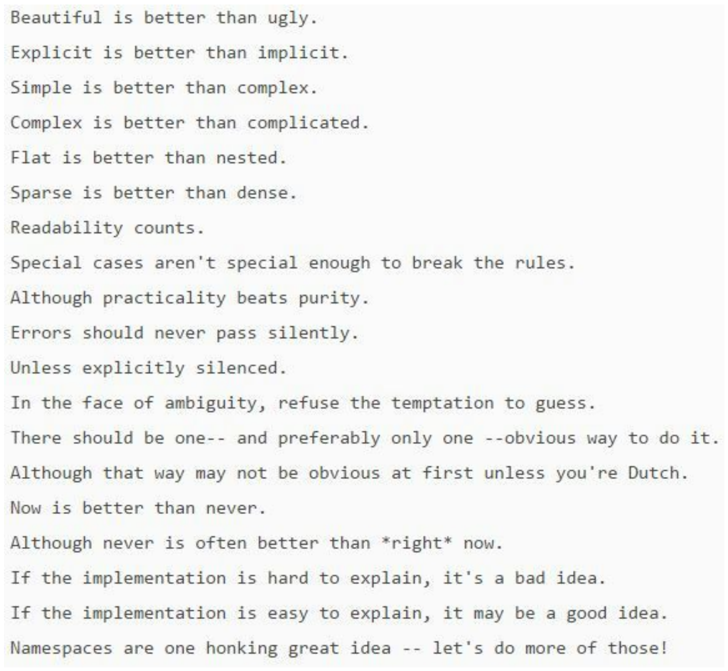
C

```c
void foo(int x)
{
    if (x == 0) {
        bar();
        baz();
    } else {
        qux(x);
        foo(x - 1);
    }
}
```

Java

```java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 2. The Zen of Python.



```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Typing Mechanism (Xiaowei Yang)

Python is dynamically typed rather than statically typed, which means all the variables do not need to be declared before they are assigned and could even change type in the middle of a program. It definitely makes it easier to use. The flexibility leads to efficient use of a developer's time. Like all other dynamic languages, an interpreter performs all type checking at run-time, instead of during a separate compilation step.[14] Whereas in C, variables be explicitly declared and give a specific type, and later use to perform static compile-time checks of the program as well as for allocating memory locations. [15] Python also has an extensive standard library, which contains more than 100 modules including their own evolving. In addition to that, there is a large supply of open source third–party modules and packages. [14]

Implementation (Xiaowei Yang)

CPython is the main implementation of Python, which is written in C. Then the virtual machine gets to compile the Python programs into intermediate byte code. [16] CPython as a source code interpreter, has a foreign function interface with many other languages including C. [17] The large standard library is actually a mixture of C and Python libraries, which is available for plenty of platforms.
The just-in-time compiler of PyPy makes a significant speed improvement over CPython. [18] PyPy with the support of the stackless mode, offers micro-threads for massive concurrency.[19]

Cython, other than CPython, is a compiled language with a foreign function interface that is obtained C/C++ routines.[20] Therefore, it has more C features like the ability to declare the types of the local variables and class attributes.

2. BNF Grammar and Rail Diagrams

This section will discuss snippets of Python's BNF grammar as well as the methods used to translate it into EBNF for use in creating a railroad diagram.

BNF Grammar

The grammar being defined in BNF appears on the left hand side of '::=' and ":" in Python's version. This symbol represents 'is defined as'. What can follow after this symbol could be a series of operators or previously defined grammar that is recursively called. This is all taken into account when creating the syntax diagram.

If we were going to look at a small piece of code from the Python BNF found [21] grammar such as

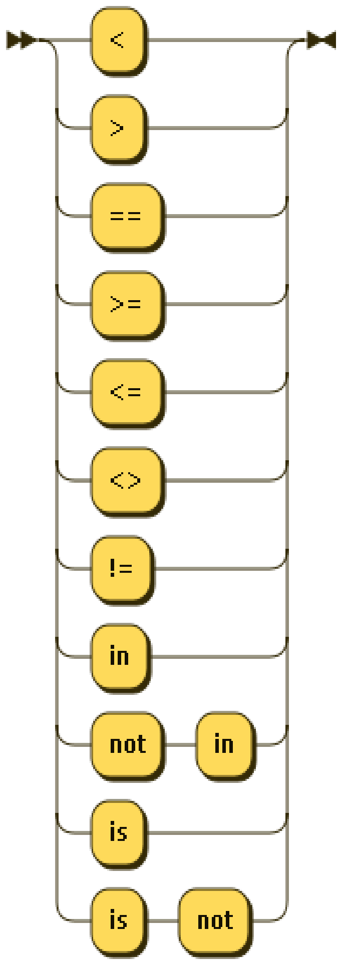
```
comp_op:  '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
```

The grammar being defined is the comparison operators followed by a '|' pipe which represents or. In plain language this would be read as: "comp_op is defined as the symbol '<' or '>' or '==' etc". On a rail diagram this can be represented by branches contained in a rounded rectangle. This is seen in *fig. 1* below. These are terminal statements, recognized by being the keywords of a language.

Terminal Statements are represented by a rounded rectangle. ▢

Non-terminal statements are represented by a square rectangle. □

fig. 1

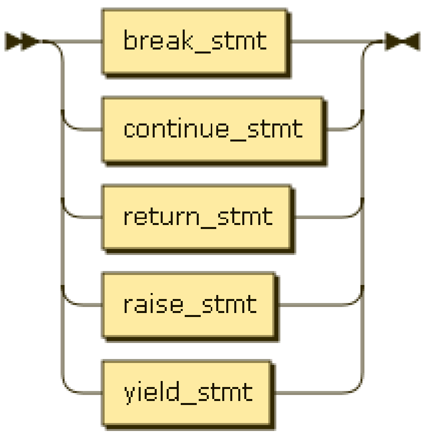


Non-terminal statements are a sequence of terminal statements joined together with ‘|’. What is outputted from the “comp_op” statement is the following diagram to the left. The arrows and lines indicate which direction that the statement originated and ended.

A more complex example would be the following snippet of Python’s BNF: [21]

flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr

fig. 2



The “flow_stmt” diagram to the right is made up of non-terminal statements defined in a similar fashion to the “comp_op” diagram. They are non-terminal and encapsulated by a rectangle because they are defined in Python’s grammar. The “flow_stmt” occurs in no specified order so each statement is self referential.

We found that Python uses it’s own variation of formal grammar, so in order to convert it into a railroad diagram we had to massage the language into EBNF, which is used by the Bottlecaps website. To do this we first transformed the assignment operators used in Python’s grammar to EBNF that the bottlecaps website above. For example,

“ : ” - > “::=”

This was found from observing the EBNF documentation[23]. Also, Python’s grammar uses meta symbols “[“ in conjunction with “]” to define optional arguments. In EBNF this changes to:

“[“ -> “(“

AND “]” -> “)?”

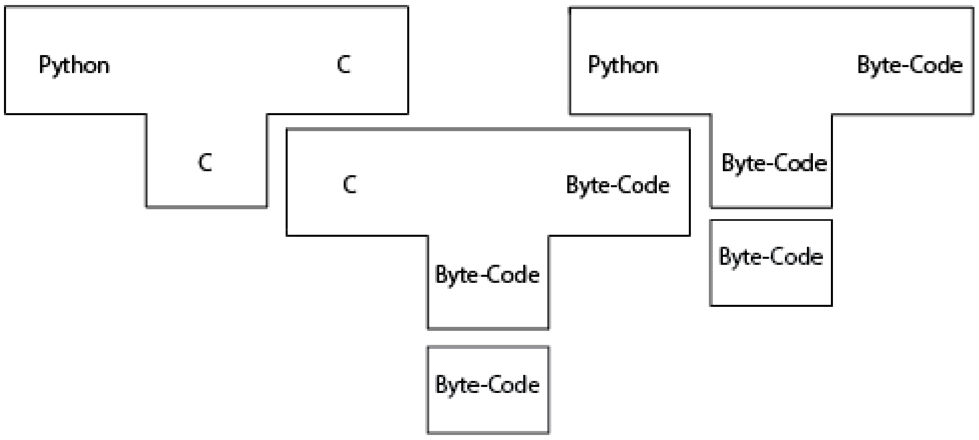
Finally, comments were changed from Python’s “#” to “/*…*/”. These three changes allowed us to translate Python’s grammar to Extended BNF grammar to create a railroad diagram.

Question 3: Compilers, Interpreters and T-Diagrams

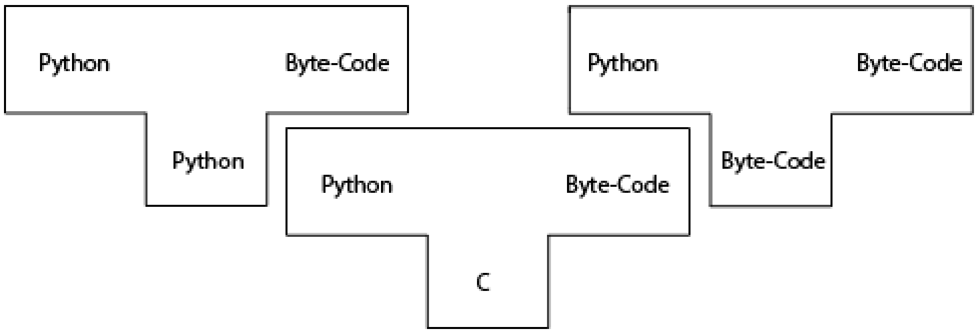
In 1991, after over a year of personally working on and developing his new scripting language, Guido Van Rossum published the first version of python code (version 0.9.0). Rossum was working at a company named CWI (Centrum voor Wiskunde en Informatica) and being inspired by the ABC programming language, he wanted to develop an interpreter for a new scripting language that would use the best features of C and ABC language family[29].

An interpreter is a kind of software package or program that executes other programs. Practically every Python implementation consists of an interpreter (as opposed to a compiler). Conclusively, this means that Python can be viewed both as a programming language and as an interpreter in itself. When you write Python programs, the Python interpreter reads your program, and carries out the instructions it contains. Conceptually, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

After being developed to include many features of C and the ABC family, Python was first implemented in C. In order to convert the Python script to machine-compatible code (byte-code), the process of bootstrapping needed to take effect[27]. It began by using the Python interpreter (written in C) to convert the python script into C compatible code. The output from this compilation would be inputted into a second system that would take C code and convert it to machine-compatible code using a machine-compatible code interpreter, in this case the machine-compatible code is called “byte-code”[30]. The resulting T-diagram to represent this bootstrap would look like this:



Once the compiler has been compiled once, it can compile itself. The following T-diagram illustrates this:



When discussing interpreters and compilers, a distinction must be made in the function and production of each system. An interpreter will take high-level code as input and executes an output. A compiler on the other hand takes high-level code and produces the object code. This code must be executed in order to produce output. Python is commonly implemented using a byte-code compiler and an interpreter[26]. As the modules and libraries are loaded, compilation is implicitly performed and required to be available at run-time (due to several language primitives). When the Python package is installed on a machine, it generates a number of components; at the very least, an interpreter and a support library. Depending on how it is used, the Python interpreter may function as an executable program, or a set of libraries linked to another program. Regardless of the Python code written, it must always be run by this interpreter.

The byte-code that is outputted by the compiler for the Python Virtual Machine typically contains the file extension .pyc (similar to Java's .class file extension)[28]. This code is completely different from the machine code generated by a C compiler for a native machine architecture. Some Python implementations including Py-Py do however consist of a JIT (just-in-time) compiler that will compile Python code into native machine code.

Although Python's standard implementation is written in C, and available for every imaginable hardware/software platform, several other implementations have become popular within the recent years. Jython is a version that runs on the JVM and has seamless Java integration. IronPython is a version for the Microsoft .NET platform that has similar integration with other languages running on .NET[28]. PyPy is an optimizing Python compiler/interpreter written in Python (still a research project, being undertaken with EU funding). There's also Stack-less Python, a variant of the C implementation that reduces reliance on the C stack for function/method calls, co-routines, continuations, and micro-threads (2Q3).

CPython

The standard form of Python is CPython. Implemented in C with C89 standard, it is an interpreter that translates programs written in Python into bytecode[31]. With open-source Python, using CPython is said to be the best for reaching the widest possible audience[28].

Historically, there were only two steps in the compilation process to byte code: the code was parsed into a syntax tree, then byte code was generated using previously generated syntax tree[26]. This method of compilation was used up to Python 2.4. In order to match the steps taken in a standard compilation process, CPython was updated to compile in the following steps:

1. Create a parse tree from the source code.
2. Change the parse tree into an Abstract Syntax Tree (AST)
3. Create a Control Flow Graph (CFG) from the AST
4. Generate bytecode from the CFG.

The resulting bytecode is then executed on the VM.

PyPy

The PyPy interpreter is a faster alternative to CPython[32]. It has a Tracing Just-In-Time (JIT) compiler, which enables Python source code to be turned into machine specific code at runtime[33].

A Tracing JIT compiler allows for optimization of programs at running time[34]. It is different from a JIT compiler in that a JIT compiler translates individual methods to bytecode, whereas a Tracing JIT compiler examines code to identify areas that are executed more frequently.

Initially PyPy was a Python interpreter that was written in Python[33]. Now, RPython is used to convert to C code, which is then compiled. RPython is short for Restricted Python, and is a small subset of the Python language. RPython starts with code objects, which are segments that define behaviour for parts of code[35]. Then, the steps in the translation from RPython to an executable are as follows:

1. The input code objects are translated into a CFG
2. An Anotator assesses the CFG to infer data types
3. The output from the Anotator is used by a typer which tranforms high-level language closer to machine language
4. Optional optimizations may be applied at this point, such as a garbage collector
5. Graphs are converted to machine code.

Stackless Python

Stackless Python is a variant of CPython which was mainly implemented in PyPy [36]. It is an interpreter that does not use the C stack, which enables it to run large concurrent programs[31]. It uses microthreads; they allow the program to be run completely by the interpreter, and they are managed by the interpreter instead of the CPU.

4. Type Construction and Type Checking(Tong Zhao, Yun-Chien Wu)

Type Construction:

Python, like any other programming language, has a few basic types, which include int, bool, and float as well as others. The Programmer is able to construct new types. The simplest example of a type a user can construct is an ordered Cartesian product[37], such as T = T1 × T2 and the reason it is considered ordered is because of the order dependency of T. This means that T' = T2 × T1 is different from T = T1 × T2 .

By using cartesian products with types we can construct a tuple which is one of the most important kinds of constructed types for python. An n-tuple is a sequence of n ordered elements, those elements may be of any type. If tuple T = T1 × T2 is the product of two types T1 and T2, then T contains as many elements as |T1| × |T2|, where |T1| and |T2| are the cardinality of the sets T1 and T2.

Users may also construct sequences, such as string, list and array. There are two important differences between list and tuple. The first is that all the elements in a list must have the same type. The second is that the type declaration does not predefine the number of elements. For arrays, the elements stored into an array are indexed. For example,

W = [u] where <a0, a1, a2,…, an>, a ∈ u.

There is also a special type of array called dictionary, also know as the associative array[38]. It maps one objects to values. Associative arrays are more common in dynamically typed languages.

Classes are a special type that contains itself[39]. In other words, a class is a type that can gets itself as a parameter. Classes in Python are not restricted structures, new methods can be added or deleted into this type.

The construction of functions in python is seen as value. Functions can map an input set into another set to receive result. For example, U -> (V1 × V2), where U is the input and (V1 × V2) is the result. Functions are considered first-class objects[40] in Python, so if you want to pass a function to another function, you can treat it the same as any other object. For example:

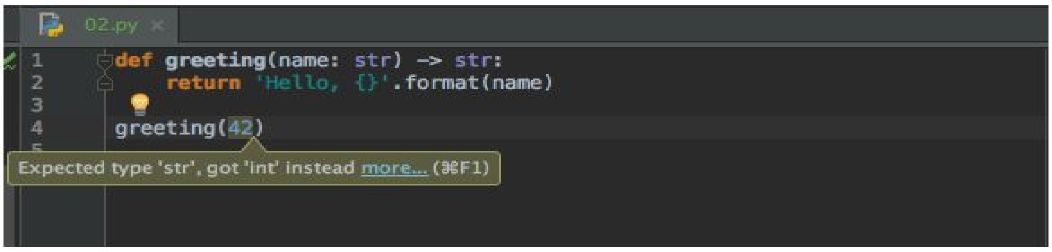
```
W = U -> V
    (U1 × U2) -> V
    U -> (V1 × V2)
```

The notation A -> B refers to the set of all function with input A and output B. Moreover, U -> V -> W is called higher-order functions. Here is another example:

```
f =  (U1 × U2) -> ( (V1 × V2) -> (W1 -> W2) )
g = f (U1 , U2)
h = g (V1, V2)
  = h (W1)
f : [u] -> [v]      which [u] as input return [v] (sequence)
```

Type Checking:

Type checking is a program analysis that verifies the type safety of a program and enforce the constraints of types, and may occur either at compile-time or run-time. Type checking done by the compiler is called static type-checking, while type checking at run-time called dynamic type checking[47]. Static type-checking allows the user to more easily find errors in their code that result from accidentally using valued of unintended types[41]. In addition, type-checking is entirely opt-in[41]. Static type checking may improve runtime efficiency since no run time type checking is needed[46]. This ensures that certain types of programming errors will be detected and reported early in development. However, it takes a long time for the compiler to check all possible type errors if the program is very large. Also, since static type checkers are necessarily conservative, the checker may disallow a program that would execute without error. Therefore, it may be less expressive[48].

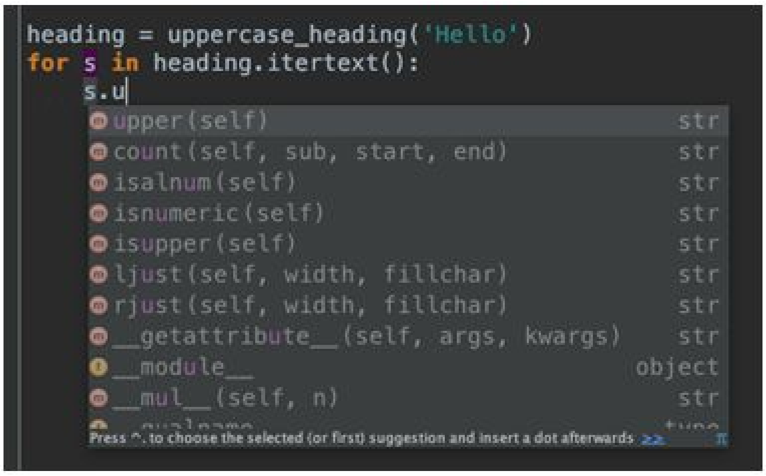


```
02.py

def greeting(name: str) -> str:
    return 'Hello, {}'.format(name)

greeting(42)
```

```
Expected type 'str', got 'int' instead more... (⌘F1)
```

Figure 1: An example of type checking [45]

Type hinting provides a way to optionally annotate the types of Python functions, arguments, and variables in a way that can be used by various tools[42]. Some programmers like help from their tools when writing code and documentation. For example, lint-like tools can spot errors before the programmer's code executes and code editors and IDEs are able to show your mistakes, while you type. Such help is particularly valuable when dealing with large code bases and when on large teams. Common mistakes can include passing the wrong type of value to a function. Using the *typing* module, the programmer can provide type hint information in the programmer's code[43]. The following code is an example:



```
heading = uppercase_heading('Hello')
for s in heading.itertext():
    s.u
```

```
upper(self)                            str
count(self, sub, start, end)           str
isalnum(self)                          str
isnumeric(self)                        str
isupper(self)                          str
ljust(self, width, fillchar)           str
rjust(self, width, fillchar)           str
__getattribute__(self, args, kwargs)   str
__module__                          object
__mul__(self, n)                       str
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>
```

Figure 2: An example of type-hinting[45]

The benefits of introducing type "hints" in Python are that:

- Develop better/faster: It can surface mistakes more quickly and go deeper than a type checker to complete the code[45].

- Help users of your code: If someone wants to use your function, type hinting helps explain and flag when they mess up[44].
- Documentation: Rather than pack argument and response type information into carefully-formatted docstrings, use something in the language[43].

The downsides of type hinting is that it forces developers to create large numbers of interfaces to keep a correct level of interoperability with other developers, it limits overriding capabilities, and tends to complexify API by introducing a lot of irrelevant method names. Furthermore, if the programmers really want to deal with bugs, they will still need to test the application. Type hinting is at the same limited and insufficient for testing an application[44].

Appendix:

Section 1:

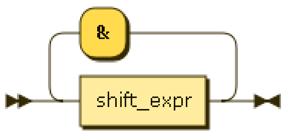


Figure 1.1 : and_expr ::= shift_expr ('&' shift_expr)*
referenced by:

- xor_expr

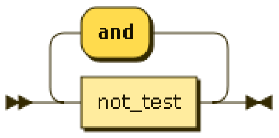


Figure 1.2 : and_test ::= not_test ('and' not_test)*
referenced by:

- or_test

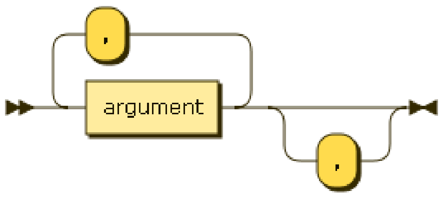


Figure 1.3 : arglist ::= argument (',' argument)* ','?
referenced by:

- classdef
- decorator

- trailer

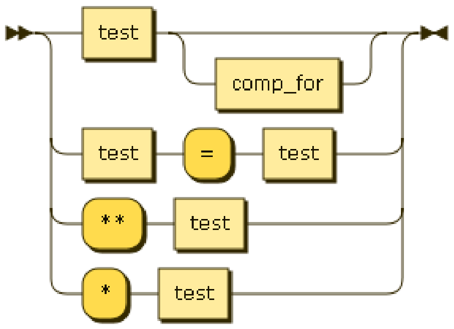


Figure 1.4 : argument ::= test comp_for?
| test '=' test
| '**' test
| '*' test
referenced by:

- arglist

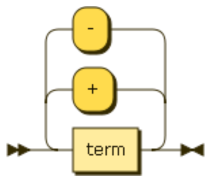


Figure 1.5 :arith_expr
::= term (('+' | '-') term)*
referenced by:

- shift_expr

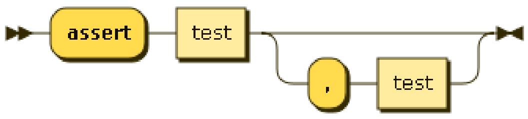


Figure 1.6 : assert_stmt
::= 'assert' test (',' test)?
referenced by:

- small_stmt

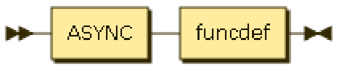


Figure 1.7 : async_funcdef
::= ASYNC funcdef
referenced by:

- decorated

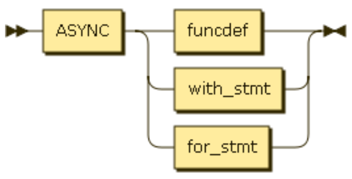


Figure 1.8 :async_stmt

::= ASYNC (funcdef | with_stmt | for_stmt)

referenced by:

- compound_stmt

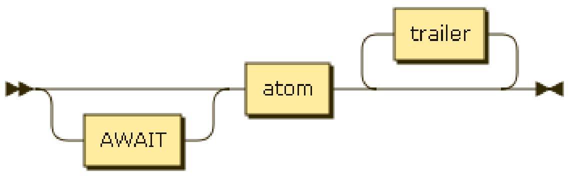


Figure 1.9 : atom_expr

::= AWAIT? atom trailer*

referenced by:

- power

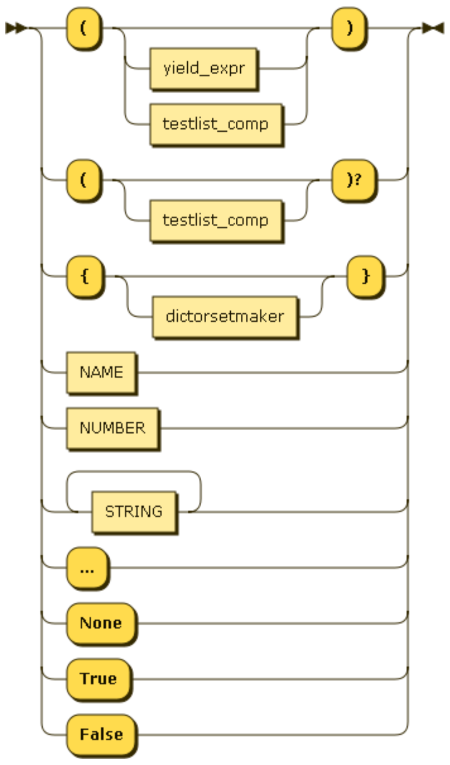


Figure 1.10 : atom ::= '(' (yield_expr | testlist_comp)? ')'

| '(' testlist_comp? ')?'

| '{' dictorsetmaker? '}'

| NAME

| NUMBER

| STRING+

| '...'
| 'None'
| 'True'
| 'False'

referenced by:

- atom_expr

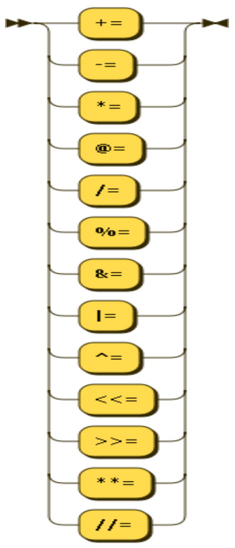


Figure 1.11 : augassign

::= '+='
| '-='
| '*='
| '@='
| '/='
| '%='
| '&='
| '|='
| '^='
| '<<='
| '>>='
| '**='
| '//='

referenced by:

- expr_stmt

Figure 1.12 : break_stmt

::= 'break'

referenced by:

- flow_stmt

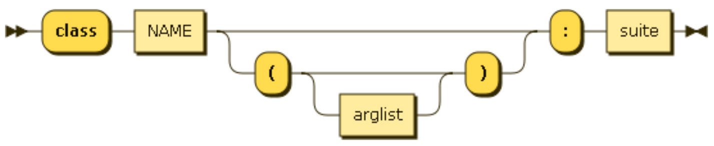


Figure 1.13 :classdef ::= 'class' NAME ('(' arglist? ')')? ':' suite
referenced by:

- compound_stmt
- decorated

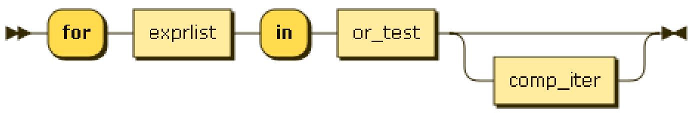


Figure 1.14 :comp_for ::= 'for' exprlist 'in' or_test comp_iter?
referenced by:

- argument
- comp_iter
- dictorsetmaker
- testlist_comp

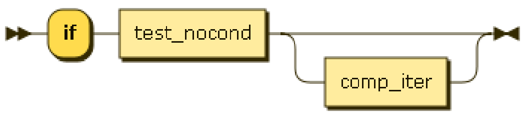


Figure 1.15 :comp_if ::= 'if' test_nocond comp_iter?
referenced by:

- comp_iter

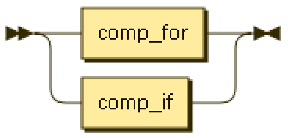


Figure 1.16 : comp_iter
::= comp_for
| comp_if
referenced by:

- comp_for
- comp_if

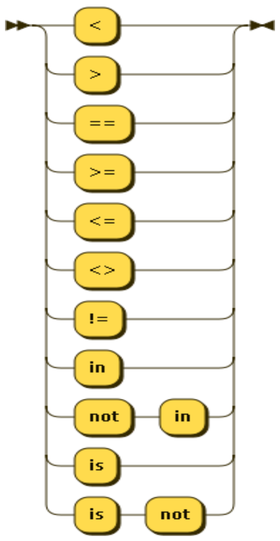


Figure 1.17 : comp_op ::= '<'
| '>'
| '=='
| '>='
| '<='
| '<>'
| '!='
| 'in'
| 'not' 'in'
| 'is'
| 'is' 'not'

referenced by:

- comparison

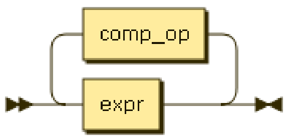


Figure 1.18 : comparison
::= expr (comp_op expr)*

referenced by:

- not_test

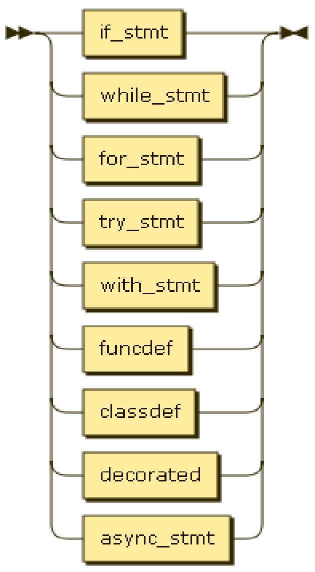


Figure 1.19 : compound_stmt

::= if_stmt
| while_stmt
| for_stmt
| try_stmt
| with_stmt
| funcdef
| classdef
| decorated
| async_stmt

referenced by:

- single_input
- stmt

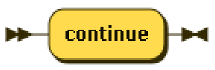


Figure 1.20 : continue_stmt

::= 'continue'

referenced by:

- flow_stmt

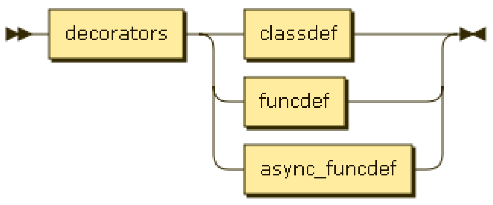


Figure 1.21 :decorated

::= decorators (classdef | funcdef | async_funcdef)

referenced by:

- compound_stmt

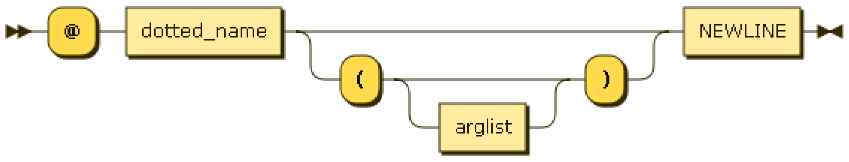


Figure 1.22 :decorator
    ::= '@' dotted_name ('(' arglist? ')')? NEWLINE

referenced by:

- decorators

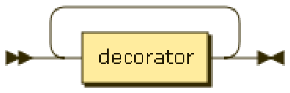


Figure 1.23 :decorators
    ::= decorator+

referenced by:

- decorated

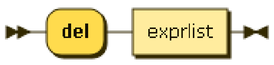


Figure 1.24 :del_stmt ::= 'del' exprlist

referenced by:

- small_stmt

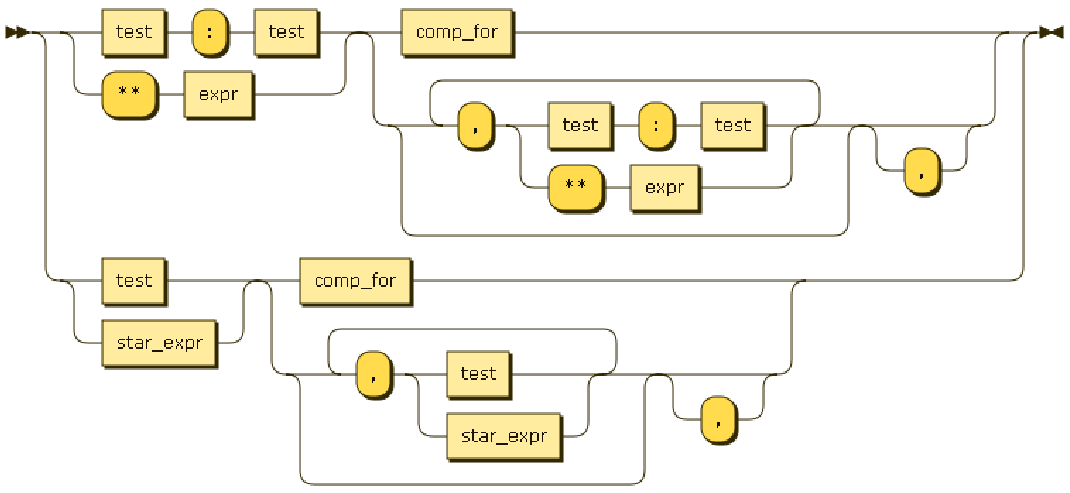


Figure 1.25 :dictorsetmaker
    ::= (test ':' test | '**' expr) (comp_for | (',' (test ':' test | '**' expr))* ','?)
      | (test | star_expr) (comp_for | (',' (test | star_expr))* ','?)

referenced by:

- atom

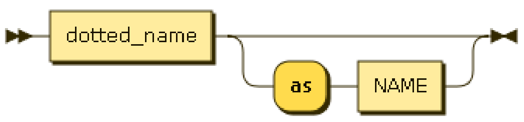


Figure 1.26 :dotted_as_name

::= dotted_name ('as' NAME)?

referenced by:

- dotted_as_names

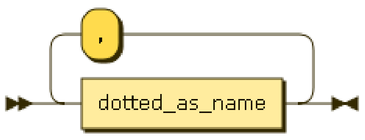


Figure 1.27 :dotted_as_names

::= dotted_as_name (',' dotted_as_name)*

referenced by:

- import_name

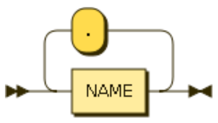


Figure 1.28 :dotted_name

::= NAME ('.' NAME)*

referenced by:

- decorator
- dotted_as_name
- import_from

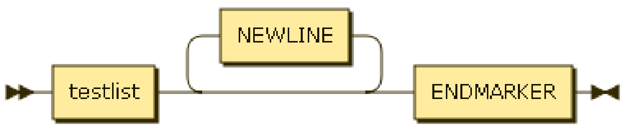


Figure 1.29 :eval_input

::= testlist NEWLINE* ENDMARKER

no references

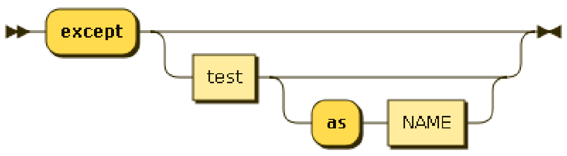


Figure 1.30 : except_clause

::= 'except' (test ('as' NAME)?)?

referenced by:

- try_stmt

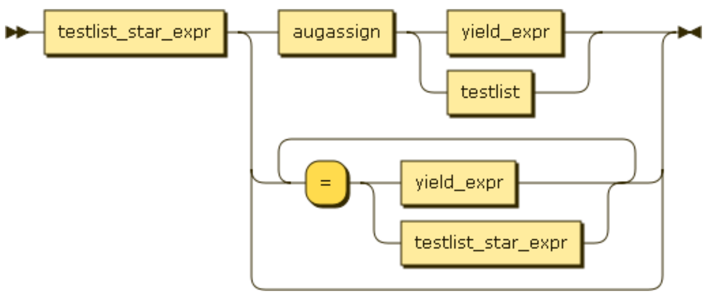


Figure 1.31 :expr_stmt
::= testlist_star_expr (augassign (yield_expr | testlist) | ('=' (yield_expr | testlist_star_expr))*)
referenced by:

- small_stmt

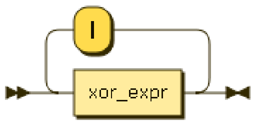


Figure 1.32 :expr ::= xor_expr ('|' xor_expr)*
referenced by:

- comparison
- dictorsetmaker
- exprlist
- star_expr
- with_item

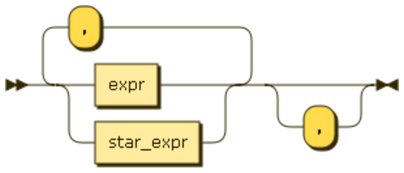


Figure 1.33 :exprlist ::= (expr | star_expr) (',' (expr | star_expr))* ','?
referenced by:

- comp_for
- del_stmt
- for_stmt

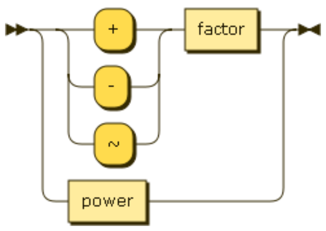


Figure 1.34 :factor ::= ('+' | '-' | '~') factor
| power

referenced by:

- factor
- power
- term

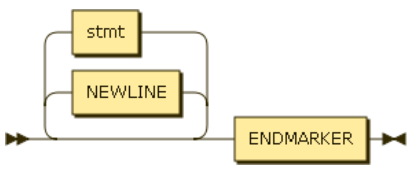


Figure 1.35 :file_input
::= (NEWLINE | stmt)* ENDMARKER
no references

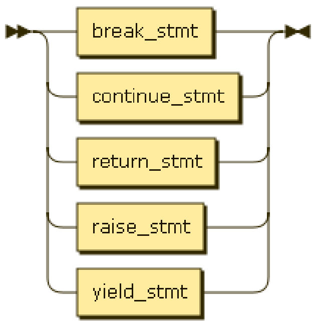


Figure 1.36 :flow_stmt
::= break_stmt
| continue_stmt
| return_stmt
| raise_stmt
| yield_stmt
referenced by:

- small_stmt

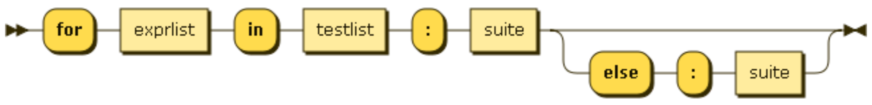


Figure 1.37 :for_stmt ::= 'for' exprlist 'in' testlist ':' suite ('else' ':' suite)?
referenced by:

- async_stmt
- compound_stmt

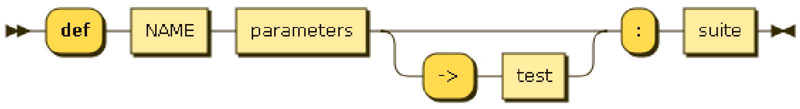


Figure 1.38 :funcdef ::= 'def' NAME parameters ('->' test)? ':' suite
referenced by:

- async_funcdef
- async_stmt
- compound_stmt
- decorated

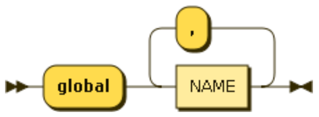


Figure 1.39 :global_stmt
::= 'global' NAME (',' NAME)*
referenced by:

- small_stmt

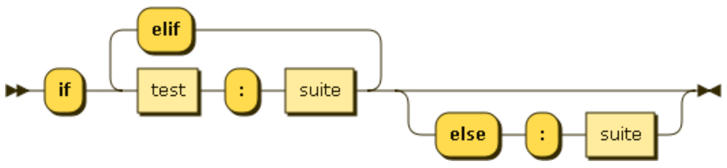


Figure 1.40 :if_stmt ::= 'if' test ':' suite ('elif' test ':' suite)* ('else' ':' suite)?
referenced by:

- compound_stmt

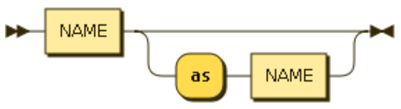


Figure 1.41 :import_as_name
::= NAME ('as' NAME)?
referenced by:

- import_as_names

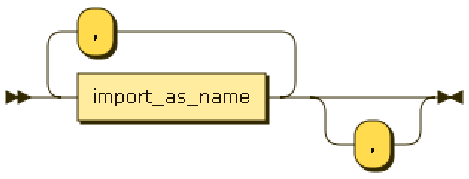


Figure 1.42 :import_as_names
::= import_as_name (',' import_as_name)* ','?
referenced by:

- import_from

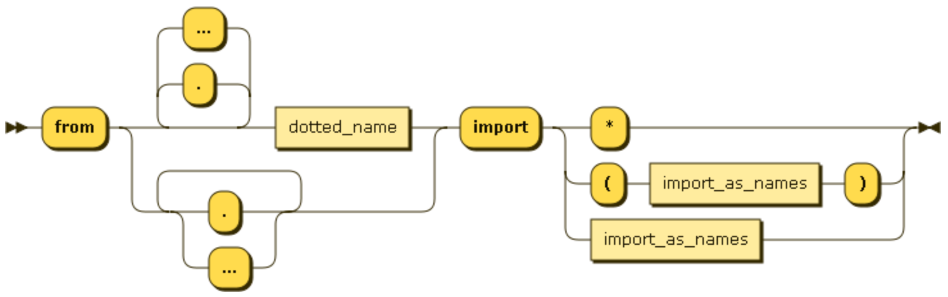


Figure 1.43 :import_from

::= 'from' (('.' | '...')* dotted_name | ('.' | '...')+) 'import' ('*' | '(' import_as_names ')' | import_as_names)

referenced by:

- import_stmt

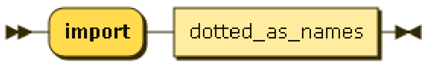


Figure 1.44 :import_name

::= 'import' dotted_as_names

referenced by:

- import_stmt

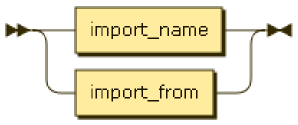


Figure 1.45 :import_stmt

::= import_name

| import_from

referenced by:

- small_stmt

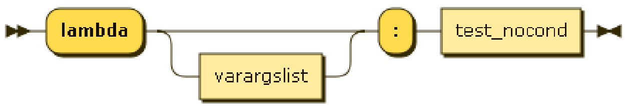


Figure 1.46 :lambdef_nocond

::= 'lambda' varargslist? ':' test_nocond

referenced by:

- test_nocond

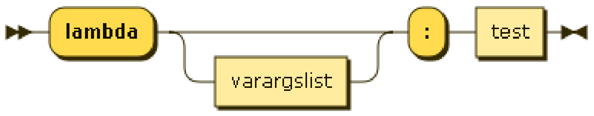


Figure 1.47 :lambdef ::= 'lambda' varargslist? ':' test

referenced by:

- test

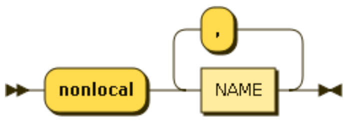


Figure 1.48 :nonlocal_stmt
::= 'nonlocal' NAME (',' NAME)*

referenced by:

- small_stmt

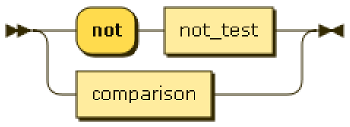


Figure 1.49 :not_test ::= 'not' not_test
| comparison

referenced by:

- and_test
- not_test

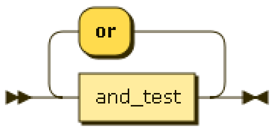


Figure 1.50 : or_test ::= and_test ('or' and_test)*

referenced by:

- comp_for
- test
- test_nocond

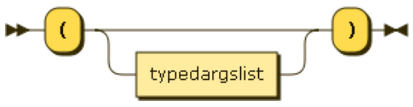


Figure 1.51 :parameters
::= '(' typedargslist? ')'

referenced by:

- funcdef

Figure 1.52 :pass_stmt
::= 'pass'

referenced by:

- small_stmt

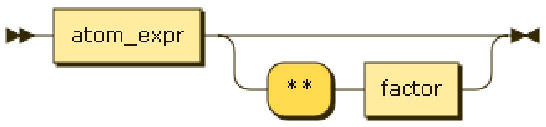


Figure 1.53 :power ::= atom_expr ('**' factor)?

referenced by:

- factor

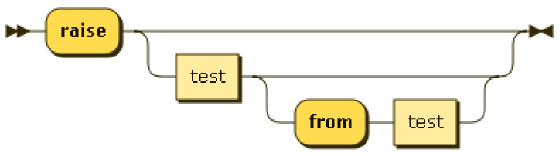


Figure 1.54 :raise_stmt

::= 'raise' (test ('from' test)?)?

referenced by:

- flow_stmt

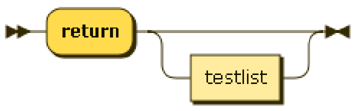


Figure 1.55 :return_stmt

::= 'return' testlist?

referenced by:

- flow_stmt

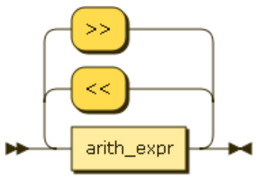


Figure 1.56 :shift_expr

::= arith_expr (('<<' | '>>') arith_expr)*

referenced by:

- and_expr

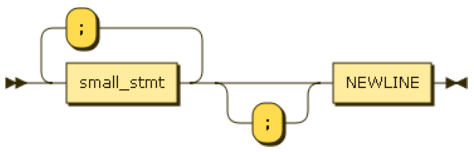


Figure 1.57 :simple_stmt

::= small_stmt (';' small_stmt)* ';'? NEWLINE

referenced by:

- single_input
- stmt
- suite

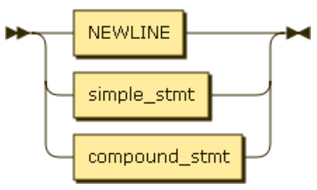


Figure 1.58 :single_input

::= NEWLINE

| simple_stmt

| compound_stmt

no references

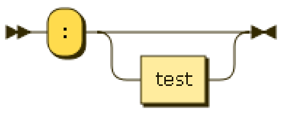


Figure 1.59 : sliceop ::= ':' test?

referenced by:

- subscript

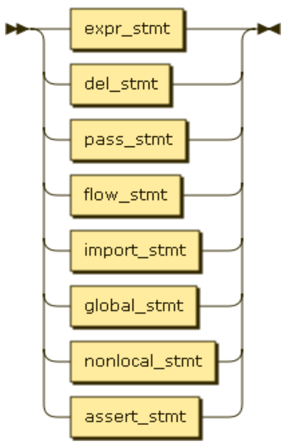


Figure 1.60 :small_stmt

::= expr_stmt

| del_stmt

| pass_stmt

| flow_stmt

| import_stmt

| global_stmt

| nonlocal_stmt

| assert_stmt

referenced by:

- simple_stmt

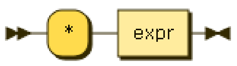


Figure 1.61 :star_expr
::= '*' expr
referenced by:

- dictorsetmaker
- exprlist
- testlist_comp
- testlist_star_expr

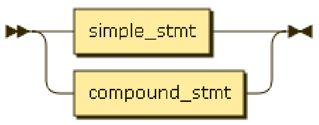


Figure 1.62 :stmt ::= simple_stmt
| compound_stmt
referenced by:

- file_input
- suite

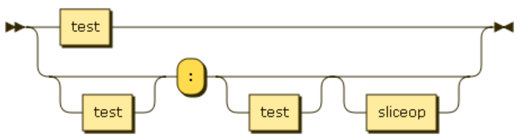


Figure 1.63 :subscript
::= test
| test? ':' test? sliceop?
referenced by:

- subscriptlist

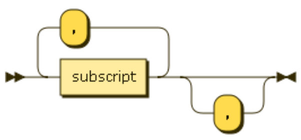


Figure 1.64 :subscriptlist
::= subscript (',' subscript)* ','?
referenced by:

- trailer

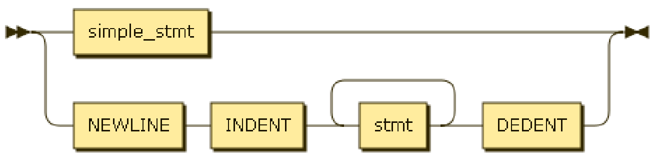


Figure 1.65 :suite ::= simple_stmt
| NEWLINE INDENT stmt+ DEDENT

referenced by:

- classdef
- for_stmt
- funcdef
- if_stmt
- try_stmt
- while_stmt
- with_stmt

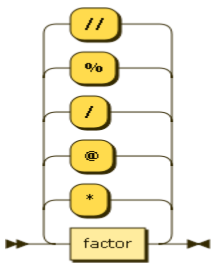


Figure 1.66 :term ::= factor (('*' | '@' | '/' | '%' | '//') factor)*

referenced by:

- arith_expr

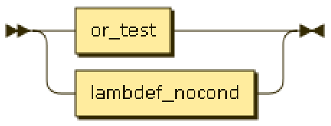


Figure 1.67 :test_nocond
::= or_test
| lambdef_nocond

referenced by:

- comp_if
- lambdef_nocond

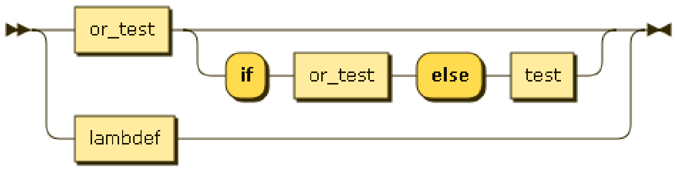


Figure 1.68 :test ::= or_test ('if' or_test 'else' test)?
| lambdef

referenced by:

- argument
- assert_stmt
- dictorsetmaker
- except_clause
- funcdef
- if_stmt

- lambdef
- raise_stmt
- sliceop
- subscript
- test
- testlist
- testlist_comp
- testlist_star_expr
- tfpdef
- typedargslist
- varargslist
- while_stmt
- with_item
- yield_arg

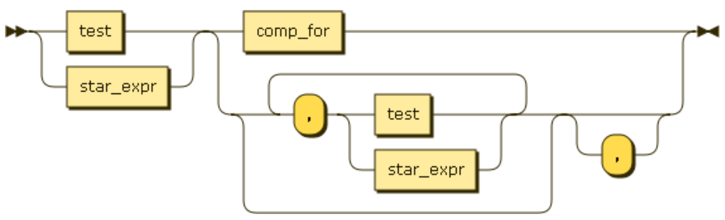


Figure 1.69 : testlist_comp

::= (test | star_expr) (comp_for | (',' (test | star_expr))* ','?)

referenced by:

- atom

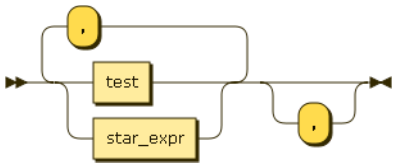


Figure 1.70 :testlist_star_expr

::= (test | star_expr) (',' (test | star_expr))* ','?

referenced by:

- expr_stmt

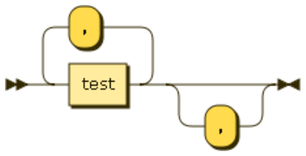


Figure 1.71 :testlist ::= test (',' test)* ','?

referenced by:

- eval_input
- expr_stmt
- for_stmt

- return_stmt
- yield_arg

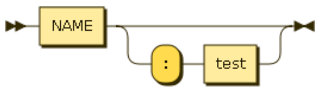


Figure 1.72 :tfpdef ::= NAME (':' test)?
referenced by:

- typedargslist

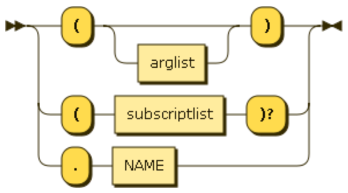


Figure 1.73 :trailer ::= '(' arglist? ')'
| '(' subscriptlist ')?'
| '.' NAME
referenced by:

- atom_expr

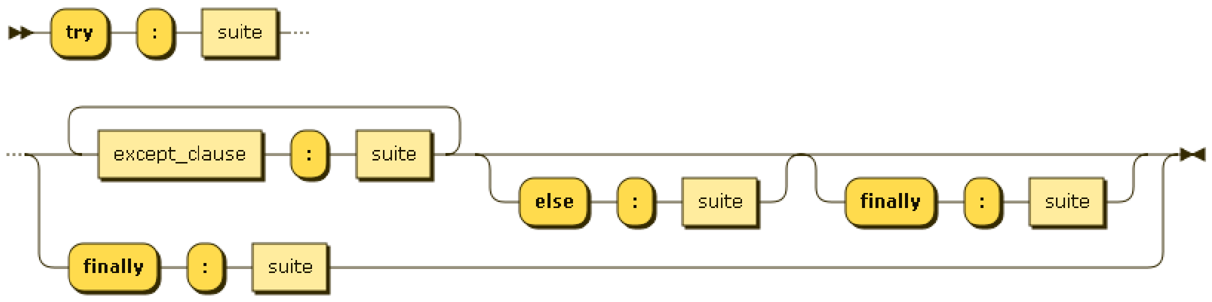


Figure 1.74 :try_stmt ::= 'try' ':' suite ((except_clause ':' suite)+ ('else' ':' suite)? ('finally' ':' suite)? | 'finally' ':' suite)
referenced by:

- compound_stmt

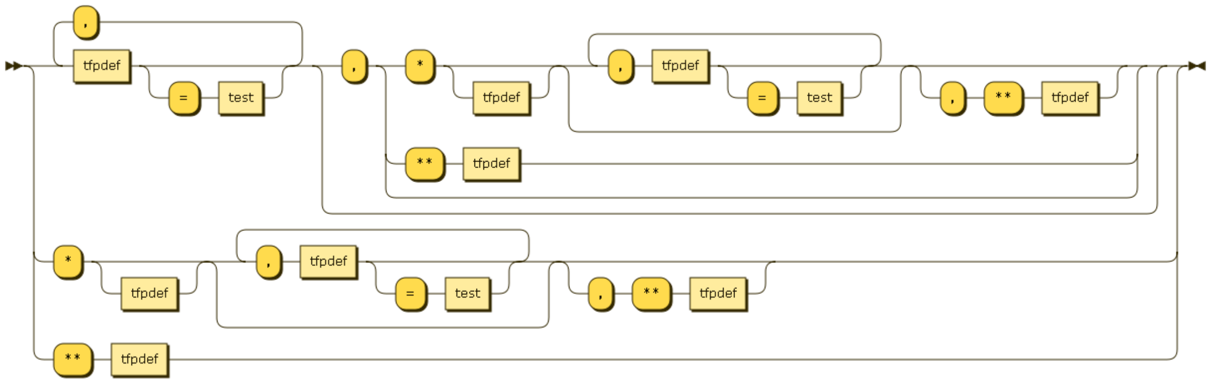


Figure 1.75 :typedargslist

::= tfpdef ('=' test)? (',' tfpdef ('=' test)?)* (',' ('*' tfpdef? (',' tfpdef ('=' test)?)* (',' '**' tfpdef)? | '**' tfpdef)?)?

| '*' tfpdef? (',' tfpdef ('=' test)?)* (',' '**' tfpdef)?

| '**' tfpdef

referenced by:

- parameters

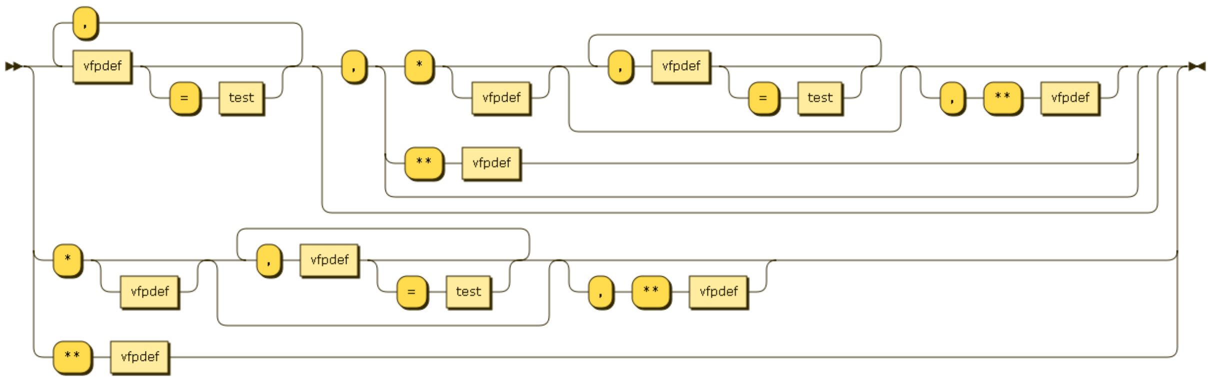


Figure 1.76 :varargslist

::= vfpdef ('=' test)? (',' vfpdef ('=' test)?)* (',' ('*' vfpdef? (',' vfpdef ('=' test)?)* (',' '**' vfpdef)? | '**' vfpdef)?)?

| '*' vfpdef? (',' vfpdef ('=' test)?)* (',' '**' vfpdef)?

| '**' vfpdef

referenced by:

- lambdef
- lambdef_nocond

Figure 1.77 :vfpdef ::= NAME

referenced by:

- varargslist

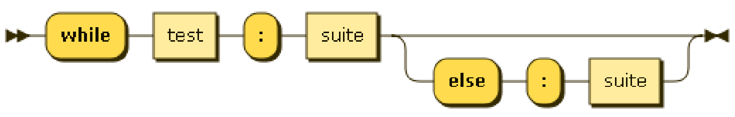


Figure 1.78 :while_stmt

::= 'while' test ':' suite ('else' ':' suite)?

referenced by:

- compound_stmt

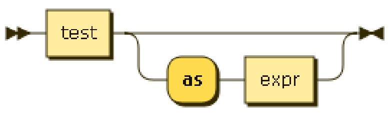


Figure 1.79 : with_item
::= test ('as' expr)?
referenced by:

- with_stmt

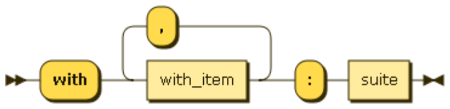


Figure 1.80 :with_stmt
::= 'with' with_item (',' with_item)* ':' suite
referenced by:

- async_stmt
- compound_stmt

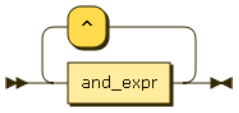


Figure 1.81 :xor_expr ::= and_expr ('^' and_expr)*
referenced by:

- expr

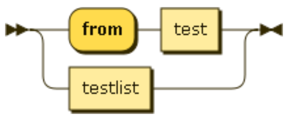


Figure 1.82 :yield_arg
::= 'from' test
| testlist
referenced by:

- yield_expr

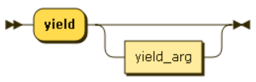


Figure 1.83 :yield_expr
::= 'yield' yield_arg?
referenced by:

- atom
- expr_stmt
- yield_stmt

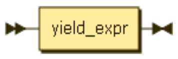


Figure 1.84 :yield_stmt
::= yield_expr

referenced by:

- flow_stmt

References:

[1]

Q1

[1]
https://docs.python.org/2/faq/general.html#why-was-python-created-in-the-first-place
[2] http://python-history.blogspot.ca/2009/01/brief-timeline-of-python.html
[3] https://docs.python.org/2/whatsnew/2.0.html
[4] https://docs.python.org/2/faq/general.html#what-is-python-good-for
[5]
https://docs.python.org/2/faq/general.html#is-python-a-good-language-for-beginning-programmers
[6] https://en.wikipedia.org/wiki/Python_syntax_and_semantics
[7] https://www.cs.purdue.edu/homes/hosking/m3/reference/syntax.html
[8] http://www.tutorialspoint.com/cprogramming/c_basic_syntax.htm
[9] https://en.wikipedia.org/wiki/ABC_(programming_language)
[10] https://en.wikipedia.org/wiki/Java_(programming_language)
[11] http://www.cs.sfu.ca/~cameron/Teaching/383/syn-sem-prag-meta.html
[12] https://www.python.org/dev/peps/pep-0020/
[13] https://en.wikipedia.org/wiki/Python_(programming_language)

[14] http://python-history.blogspot.ca/2009/01/introduction-and-overview.html

[15] https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/
[16] http://www.troeger.eu/files/teaching/pythonvm08.pdf
[17] https://www.quora.com/What-is-CPython
[18] http://morepypy.blogspot.be/2012/06/stm-with-threads.html

[19] http://pypy.org/
[20] http://cython.org/
[21] https://docs.python.org/3/reference/grammar.html
[22] http://www.bottlecaps.de/rr/ui
[23] https://www.w3.org/TR/xquery/#EBNFNotation
[24] http://matt.might.net/articles/grammars-bnf-ebnf/
[25]
http://www-01.ibm.com/support/knowledgecenter/SSB27U_6.2.0/com.ibm.zvm.v620.dm
sb3/syntax.htm%23syntax
[26] https://docs.python.org/devguide/compiler.html
[27] https://en.wikipedia.org/wiki/Python_%28programming_language%29#Implementations
[28] http://docs.python-guide.org/en/latest/starting/which-python/
[29] http://groups.engin.umd.umich.edu/CIS/course.des/cis400/python/python.html
[30]
https://connex.csc.uvic.ca/access/content/group/26fa06c7-ba9d-440d-9d2f-588830dd7fd2Fr
ee%20Books/Compiler%20Construction%20and%20Bootstrapping.pdf
[31] https://en.wikipedia.org/wiki/Python_(programming_language)
[32] http://pypy.readthedocs.org/en/latest/release-1.2.0.html
[33] https://en.wikipedia.org/wiki/PyPy#RPython
[34] https://en.wikipedia.org/wiki/Tracing_just-in-time_compilation
[35] https://rpython.readthedocs.org/en/latest/architecture.html#architecture
[36] https://en.wikipedia.org/wiki/Stackless_Python
Q4

[37]: https://people.rit.edu/blbgse/pythonNotes/itertools.html
[38]: https://en.wikipedia.org/wiki/Associative_array
[39]: http://www.tutorialspoint.com/python/python_classes_objects.htm
[40]: https://en.wikipedia.org/wiki/First-class_function
[a]https://ericjmritz.name/2015/10/08/static-type-checking-in-python-using-mypy/
[b] http://lwn.net/Articles/650904/
[c]http://blog.jetbrains.com/pycharm/2015/11/python-3-5-type-hinting-in-pycharm-5/
[d]http://radify.io/blog/type-hinting-in-php-good-or-bad-practice/
[e]http://blog.pirx.ru/media/files/2015/type-hinting-talk/type-hinting.html#8
[f]http://www.alorelang.org/doc/typeoverview.html
[g]http://digital.cs.usu.edu/~allan/Compilers/Notes/TypeChecking.pdf
[h]http://web.cs.mun.ca/~ulf/pld/s/ts-why.html