Table of Contents

Introduction **2**

Videos **3**

1 Statements **4**

2 Interpreting a Python statement **5**

3 Tokens **6**

3.1 Keywords 7

3.2 Identifiers 8

3.3 Literals 9

3.3.1 Integer Literals 9

3.3.2 Floating Point Literals, or Float Literals 9

3.3.3 String Literals 10

3.4 Delimiter Tokens 11

3.5 Operator Tokens 12

3.6 INDENT and DEDENT Tokens 13

3.7 NEWLINE tokens 14

4 Picking the longest token **15**

5 Examples **16**

Introduction

This document describes the first step of how the Python interpreter parses your code, also known as "lexical analysis".

Take this paragraph you're reading right now. Your brain automatically splits this sequence of letters into words using the blank spaces. It also takes punctuation and symbols into account.

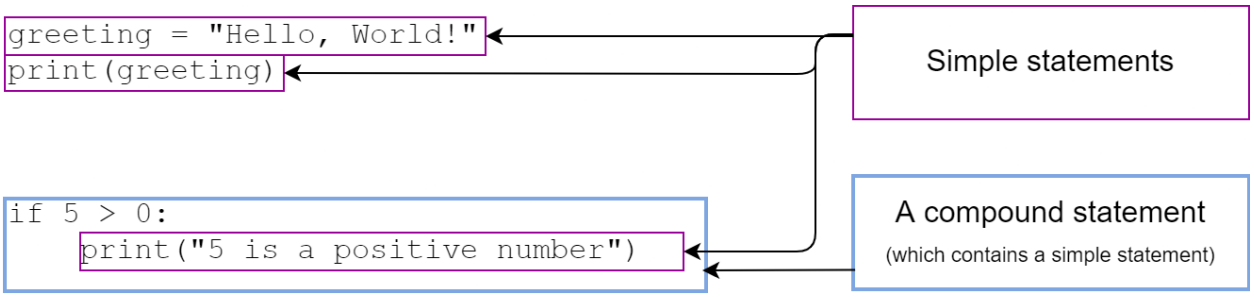
A similar process needs to occur in Python. A Python program file, like any text file, appears on a computer as a long sequence of characters. Python needs to be able to group and separate these characters into word-like sequences and recognize symbols as well. This document will describe this process at a relatively high level.

Videos

- The logical *and* operator (0:00 - 4:11)
    - Covers when and how to use the *and* operator, and includes an example
- The logical *or* operator (0:00 - 4:08)
    - Covers when and how to use the *or* operator, and includes an example
- The *not* operator (0:00 - 2:57)
    - Covers how to use the *not* operator and includes an example
- Comparison operators (0:00 - 3:59)
    - Covers the different types of comparison operators, how to use them, and includes examples

1 Statements

A Python program is composed of statements. In most cases, a statement is composed of one line of code. We call these "simple statements". Later, we will see types of statements that span multiple lines, which are called "compound statements".



2 Interpreting a Python statement

For Python to understand your code, it first needs to be able to break up a sequence of characters into so-called *tokens*, which are the smallest meaningful units in a program. You might think of these tokens as the "words", "punctuation" and "symbols" of the language.

The next step for Python is to check for syntax errors. For instance, `print("Hello"` will produce a syntax error because there is no closing parenthesis. *This process is not covered in this document.*

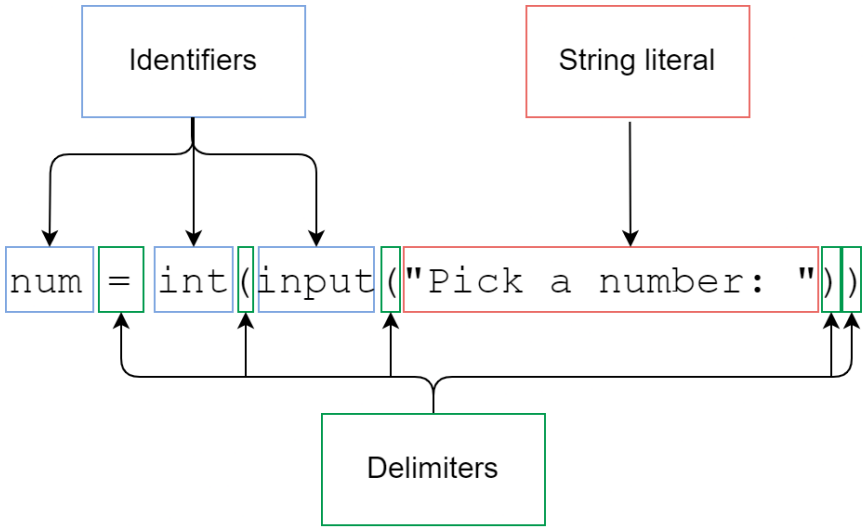
Again, we are only concerned with how Python breaks up a sequence of characters into tokens, also known as tokenization. For example, in English, I could write gibberish -- Suf dewo loedi -- and you would still be able to break this up into words. Similarly, you can feed Python nonsensical code (you'll probably do this a lot) and it will still break it up into tokens. It will just spit out an error further along, when it tries to understand what you've written.

3 Tokens

A token is the smallest meaningful unit in a Python program. As Python reads through your program line-by-line, it will decompose each statement into tokens. Tokens are composed of sequences of characters.

In Python, there are several categories of tokens: keywords, identifiers, literals, operators, and delimiters. In addition, Python can also parse indentation levels and line breaks as tokens. You will need to be able to classify tokens into one of these 5 categories.

Here is a quick example of tokenization (breaking code into tokens) of a simple statement; the following sections will discuss each category of token in more detail.



Finally, a token in Python can only belong to one category. While a word in English can be both a noun and a verb, a token in Python belongs to just one.

The following subsections will describe each of the 5 categories. As you read them, keep in mind that Python classifies each token based on what characters they contain, paying no attention to the purpose (or meaning) of the token.

For a detailed technical document on the process, you can consult the Python documentation: [2. Lexical analysis — Python 3.9.5 documentation]

3.1 Keywords

Keywords are tokens in Python that are reserved for specific purposes by the language.

Here is a complete list of keywords in Python 3:

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

As a group, keywords do not have any common role. They can act as delimiters (such as the `if` keyword), reference specific objects in memory (these are the `False`, `True`, and `None` keywords), or as operators (such as `and` and `or`).

3.2 Identifiers

An identifier:

1. Begins with a letter, or an underscore.
2. Optionally, continues with letters, digits, or underscores.
3. Cannot be a Python keyword token

If we only had the first two rules, then any of the keywords, as they are composed of just letters, would be considered identifiers as well.

Identifier tokens are used to refer to objects in memory. You might use them in your code to refer to objects you've created, or you might use them to refer to built-in or imported objects. For instance, print is an identifier that is normally associated with the built-in function that prints things to the screen.

Be careful with False, True, and None. While these also refer to objects in memory, since they are keywords, they are not identifier tokens (the definition above explicitly excludes keywords from the category of identifier tokens).

Keep in mind that in Python, uppercase and lowercase letters are considered different characters. For example, `pi` and `Pi` are two different identifiers.

Here are some examples of valid and invalid identifiers:

Identifiers	**Not identifiers**	
`hello`	`if`	`(keyword token)`
`print`	`True`	`(keyword token)`
`Import`	`3letters`	`(begins with digit)`
`del_`	`my-word`	`(contains -)`
`_num`	`random+number`	`(contains +)`

3.3 Literals

Literals are used to represent constant, fixed values in your program. For instance, the number 2 will denote that number and never anything else.

For the most part, there are three kinds of literals you'll come across: integer literals, floating point literals, and string literals.

3.3.1 Integer Literals

Integer literal tokens are used to represent non-negative whole numbers in your program.

Integer literals:

1. Are composed of one or more digits and no other characters

Note that while negative integers exist in mathematics (and as objects in memory with Python), a negative integer, such as -2, when written in this way in Python code, is not a single token, but consists of two tokens, an operator token (-) followed by an integer token (2).

Integer literals	**Not integer literals**

`123`
`5`
`7`
	`-5` (contains a -)
	`7.0` (contains a .)
	`3.` (contains a .)
	`+30` (contains a +)

3.3.2 Floating Point Literals, or Float Literals

A floating point literal token is used to represent a non-negative rational number. That is, a number with a fractional component.

Float literals:

1. Contain exactly one dot
2. Contain one or more digits
3. Contain no other characters

Again, as with integers, while negative floating point numbers exist as objects in code, there are no negative floating point literal tokens.

Float literals	Not float literals

`3.`
`4.0`
`91.27912`
`.5123`
	`1.2.3` (contains two .)
	`-5.3` (contains a -)
	`+8.` (contains a +)

3.3.3 String Literals

Strings are sequences of characters surrounded by single quotation marks (`'`), or surrounded by double quotation marks (`"`). Note that you cannot start your string literal with one kind of quotation mark and end it with the other kind. You can however, use one kind of quotation mark to indicate a string literal and use the other kind of quotation mark as part of the string that you want to represent, which might come in handy when your string contains quotation marks.

String literals:

1. Begin with one single quote (`'`), or one double quote (`"`)
2. Continue with any kind of character except the quote it began with. It cannot contain newline characters (line breaks).
3. End with the same type of quote it began with.

For example,

```
"hello world"
'René Descartes said, "I think, therefore I am."'
```

3.4 Delimiter Tokens

Delimiters come from the following table:

(	)	[	]	{	}	
,	:	.	;	@	=	->
+=	-=	*=	/=	//=	%=	@=
&=	|=	^=	>>=	<<=	**=	

Delimiters are special characters whose function can vary depending on context. Just like the period in English can serve multiple purposes, so can delimiters in Python.

Take the parentheses -- `(` and `)` -- for example. They can be used to surround expressions, and are used in function calls, among other uses.

3.5 Operator Tokens

Operator tokens come from the following table:

`+`	`-`	`*`	`**`	`/`	`//`	`%`	`@`
`<<`	`>>`	`&`	`|`	`^`	`~`	`:=`	
`<`	`>`	`<=`	`>=`	`==`	`!=`		

Operator tokens generally are used to refer to operations on objects. Remember that we are purely classifying symbols in this process, not the meaning of them. There are operations in Python that are not represented by operator tokens. Here are some examples:

- for performing the logical operations conjunction and disjunction, the keyword tokens `and` and `or` are used, respectively.
- for the subscription operation, the delimiter tokens, `[` and `]`, are used.

3.6 INDENT and DEDENT Tokens

Python uses the indentation level of a line to determine the structure of the code. These are mainly used in compound statements. In a compound statement. when the indentation level *changes*, Python will parse an INDENT or a DEDENT token.

For example:

```
if 5 > 4:                                   (1)
    print("5 is greater than 4")            (2)
if 4 > 5:                                   (3)
    print("This is unexpected...")          (4)
    print("Something is gravely wrong.")    (5)
```

The second line with the `print` statement is indented one level compared to the line with the `if` keyword. Thus, there will be one INDENT token parsed before the `print` identifier.

The third line, containing the second `if` keyword, is "unindented" one level from the previous line. Thus, there will be a DEDENT token here.

Between the `print` statements on the 4th and 5th lines, there will be neither an INDENT nor a DEDENT token because the indentation level hasn't *changed*.

There can be multiple levels of indentation. Indentation levels can only happen one at a time, but lines can be dedented multiple levels at the same time.
For example:

```
if 4 > 5:                                   (1)
    if 3 > 5:                               (2)
        print("This is unexpected...")      (3)
print("Finished testing")                   (4)
```

Now, there will be an INDENT token before the `if` keyword on line 2, and the `print` identifier on line 3. Notice, however, that the 4th line is unindented to the same level as line 1. There will be two DEDENT tokens before the `print` identifier on line 4.

3.7 NEWLINE tokens

Finally, there is also a NEWLINE token. Generally, a line break is used to separate statements.

```
print("Hello")
print("World")
```

The line break between the two print statements is represented by a NEWLINE token.

4 Picking the longest token

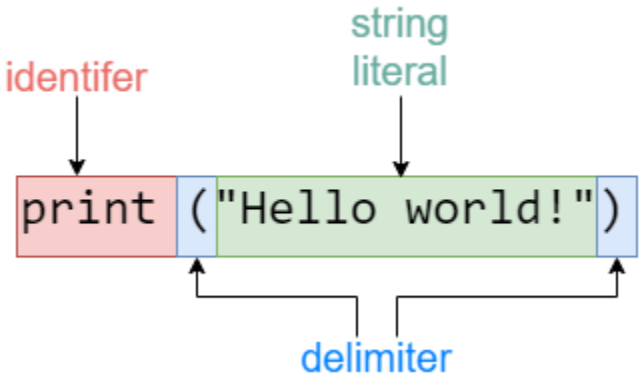
Python will read left-to-right, and choose the longest possible token. Whitespace can be used to manually separate tokens. For example, `1.2.3` will be parsed as a `1.2` literal float and a `.3` literal float. On the other hand, `1. 2.3` will produce a `1.` literal float and a `2.3` literal float.

Similarly, `==` will be parsed as the `==` operator token, but `= =` will be parsed as two `=` delimiter tokens.

5 Examples

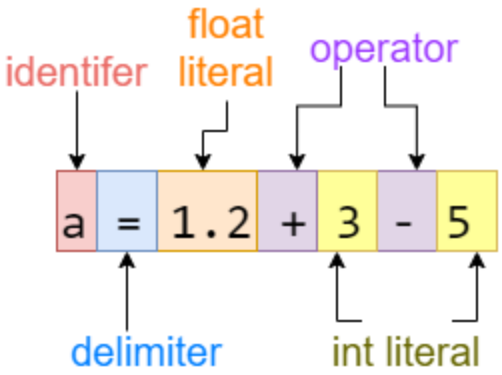
Code	Tokens	Token Kind

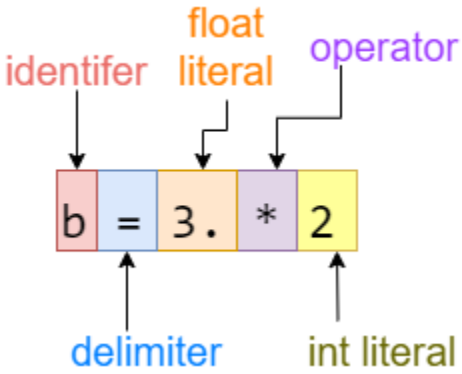
`print("Hello world!")`	`print`
	`(`
	`"Hello world!"`
	`)`
		`Identifier`
		`Delimiter`
		`String literal`
		`Delimiter`

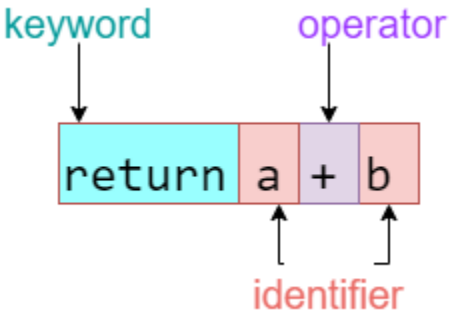


Code	Tokens	Token Kind

`a = 1.2 + 3 - 5`
`b = 3.*2`
`return a+b`
	`a`
	`=`
	`1.2`
	`+`
	`3`
	`-`
	`5`
	
	`b`
	`=`
	`3.`
	`*`
	`2`
	
	`return`
	`a`
	`+`
	`b`
		`Identifier`
		`Delimiter`
		`Float literal`
		`Operator`
		`Integer literal`
		`Operator`
		`Integer literal`
		`NEWLINE`
		`Identifier`
		`Delimiter`
		`Float literal`
		`Operator`
		`Integer literal`
		`NEWLINE`
		`Keyword`
		`Identifier`
		`Operator`
		`Identifier`

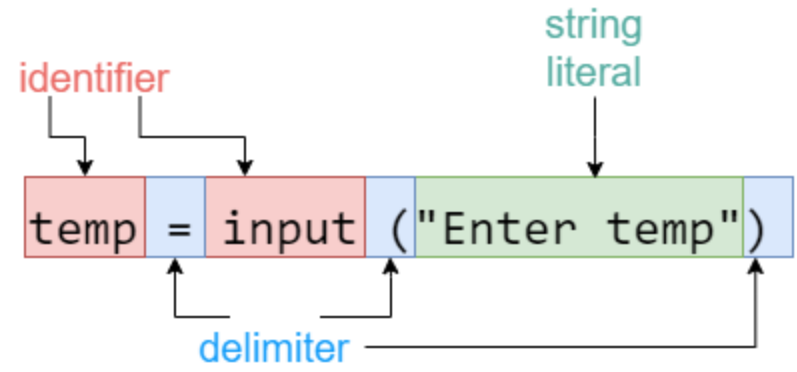

identifer
float literal
operator
a = 1.2 + 3 - 5
delimiter
int literal

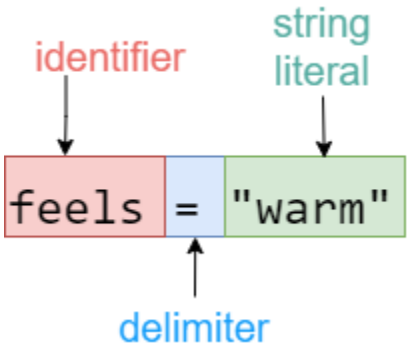

identifer
float literal
operator
b = 3. * 2
delimiter
int literal

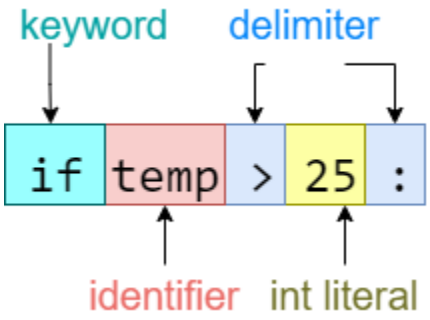

keyword
operator
return a + b
identifier

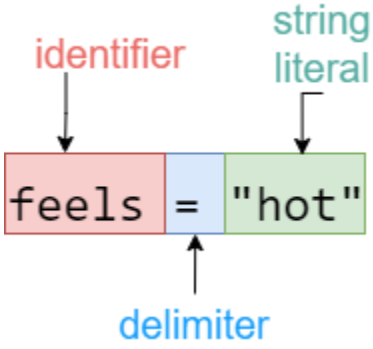
Code	Tokens	Token Kind

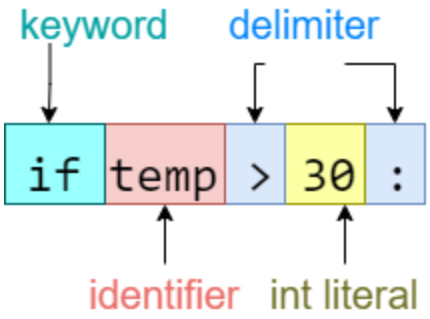
temp = input("Enter temp")
feels = "warm"
if temp > 25:
    feels = "hot"
    if temp > 30:
        feels = "too hot"
print(feels)
	temp
	=
	input
	(
	"Enter temp"
	)
	
	feels
	=
	"warm"
	
	if
	temp
	>
	25
	:
	
	feels
	=
	"hot"
	
	if
	temp
	>
	30
	:
	
	feels
	=
	"too hot"
	
	print
	(
	feels
	)
		Identifier
		Delimiter
		Identifier
		Delimiter
		String literal
		Delimiter
		NEWLINE
		Identifier
		Delimiter
		String literal
		NEWLINE
		Keyword
		Identifier
		Operator
		Integer literal
		Delimiter
		NEWLINE
		INDENT
		Identifier
		Delimiter
		String literal
		NEWLINE
		Keyword
		Identifier
		Operator
		String literal
		Delimiter
		NEWLINE
		INDENT
		Identifier
		Delimiter
		String literal
		NEWLINE
		DEDENT
		DEDENT
		Identifier
		Delimiter
		Identifier
		Delimiter

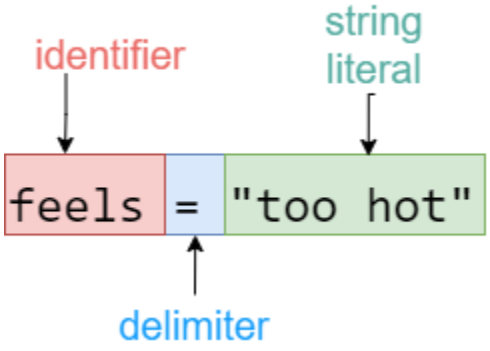

identifier
string literal
temp = input ("Enter temp")
delimiter

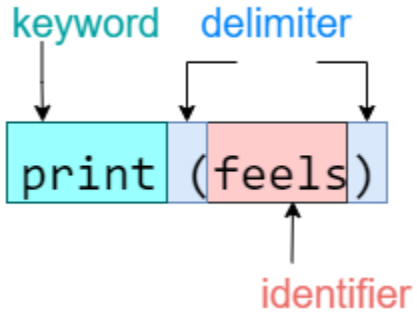

identifier
string literal
feels = "warm"
delimiter


keyword
delimiter
if temp > 25 :
identifier
int literal


identifier
string literal
feels = "hot"
delimiter


keyword
delimiter
if temp > 30 :
identifier
int literal


identifier
string literal
feels = "too hot"
delimiter


keyword
delimiter
print (feels)
identifier