

Python Evaluation Rules

UW CSE 160

<http://tinyurl.com/dataprogramming>

Michael Ernst and Isaac Reynolds

mernst@cs.washington.edu

August 2, 2016

Contents

1	Introduction	2
1.1	The Structure of a Python Program	2
1.2	How to Execute a Python Program	2
2	Literal Expressions	3
3	Operator Expressions	3
3.1	Binary Expressions	3
3.2	Compound Expressions	4
3.3	Unary Expressions	5
4	Variables	6
4.1	Variable Access Expressions	7
4.2	Variable Assignment Statements	7
5	If Statements	9
5.1	Rules for Evaluation	10
5.2	Examples	10
6	Data structures: Lists, Tuples, Sets, and Dictionaries	13
6.1	Constructor Expressions	14
6.2	Data Structure Access Expressions	16
6.3	Data Structure Assignment Statements	18
6.4	Sequence Slice Access and Assignment Expressions	20
6.5	del Statements	23
7	Loop Statements	24
7.1	for Loop Statements	24
7.2	while Loop Statements	28
7.3	break Statements	30
7.4	continue Statements	31
7.5	Comprehension Expressions	32

8	Functions	34
8.1	Function Definition Statements	34
8.2	Variable Access Expressions, Refined	36
8.3	Variable Assignment Statements	37
8.4	Function Call Expressions	40

1 Introduction

This document presents step-by-step rules for executing a Python program. A skilled Python programmer uses these rules to reason about the effects of Python code. This document will enable you to program more efficiently and with less confusion.

We wrote this document as a reaction to the vague English descriptions in many textbooks and websites. If you have only a fuzzy notion of what Python does, you will struggle to write correct code, debug incorrect code, and read unfamiliar code. This document might take some effort to understand on first reading, but overall it will save you time and avoid frustration.

This document applies to both Python 2 and Python 3. It does not cover every possible Python expression, but does cover the most frequently-used ones.

1.1 The Structure of a Python Program

A Python **program** is a sequence of statements. Python executes this sequence of statements in a specific, consistent, and predictable order.

A Python **statement** contains zero or more expressions. A statement typically has a side effect such as printing output, computing a useful value, or changing which statement is executed next.

A Python **expression** describes a computation, or operation, performed on data. For example, the arithmetic expression `2+1` describes the operation of adding 1 to 2. An expression may contain sub-expressions — the expression `2+1` contains the sub-expressions `2` and `1`.

An expression is some text a programmer writes, and a **value** is Python’s internal representation of a piece of data. Evaluating an expression computes a Python **value**. This means that the Python expression `2` is different from the value `2`. This document uses **typewriter font** for statements and expressions and **sans serif font** for values.

1.2 How to Execute a Python Program

Python executes a **program** by executing the program’s statements one by one until there are no more statements left to execute. In general, Python executes statements from top to bottom.

Python executes a **statement** by evaluating its expressions to values one by one, then performing some operation on those values.

Python evaluates an **expression** by first evaluating its sub-expressions, then performing an operation on the values. Notice that each sub-expression might have its own sub-sub-expressions, so this process might repeat several times. However, this process of dividing and evaluating always terminates because the expressions become smaller at each step until they reach some base expression.

For example, to evaluate `2*10 + 6/3`, Python first evaluates `2*10` to the value `20`, then evaluates `6/3` to the value `2`, then adds the two to get `22`. Note that in order to evaluate one expression, Python evaluates several smaller expressions (such as `2*10`). Furthermore, to evaluate `2*10`, Python evaluates the expression `2` to the value `2`, and so forth. The value of a literal expression such as `2` is the corresponding value, so this is where Python stops dividing into sub-expressions.

The remainder of this document gives evaluation rules for Python expressions and statements.

The general approach is *rewriting*. Given a statement or expression, each individual rule does a tiny bit of work that simplifies the statement or expression to an equivalent but shorter or simpler version, until

there is no more work to do and you have executed the whole thing. The general idea is to break up an imposing task into bite-sized pieces. Evaluation of any program proceeds by small, simple steps, and by understanding these you can understand the execution of the whole program.

Comments and corrections to this document are welcome; send them to mernst@cs.washington.edu.

2 Literal Expressions

A literal expression evaluates to the value it represents. Here are some examples of literal expressions and the values to which they evaluate:

```
17           ⇒ 17
'this is some text' ⇒ "this is some text"
8.125        ⇒ 8.125
True         ⇒ True
```

This document uses \Rightarrow to show an expression (on the left) and the value to which it evaluates (on the right).

3 Operator Expressions

3.1 Binary Expressions

A binary expression consists of a binary operator applied to two operand expressions. A binary operator is an operator that takes two arguments (for example, $+$ or $/$). Here are some examples of binary arithmetic expressions, each of which evaluates to a number:

```
2 * 5   ⇒ 10
14 + 8   ⇒ 22
```

Here are some examples of binary Boolean expressions, each of which evaluates to a Boolean (True or False):

```
6 == 7   ⇒ False
0 < 5    ⇒ True
True and False ⇒ False
```

Some expressions don't evaluate to numbers or Booleans. For instance, applying the $+$ operator to two string values evaluates to the concatenation of the two strings:

```
'have a ' + 'very good day' ⇒ "have a very good day"
```

In general, a binary expression has the form:

```
EXPR BIN_OP EXPR
```

3.1.1 Rules for Evaluation

To evaluate a binary expression to a value,

1. Evaluate the left operand (which is an expression) to a value and replace that operand expression with that value.
2. Evaluate the right operand (which is an expression) to a value and replace that operand expression with that value.
3. Apply *BIN_OP* to the two resultant values, obtaining the value of the binary expression. Replace the entire binary expression with this value.

3.1.2 Examples

Below are some examples of evaluating binary expressions. Each example starts with a binary expression and shows each step of the evaluation, ending with the value to which the binary expression evaluates. The underlined part of each expression is the part that is evaluated next.

Remember that expressions are in **typewriter** font and values are in **sans serif** font.

1. $\underline{2} * 5$
 $2 * \underline{5}$
 $\underline{2} * \underline{5}$
 10
2. $\underline{14} + 8$
 $14 + \underline{8}$
 $\underline{14} + \underline{8}$
 22
3. $\underline{\text{True}}$ and $\underline{\text{False}}$
 True and $\underline{\text{False}}$
 $\underline{\text{True}}$ and $\underline{\text{False}}$
 False
4. $\underline{\text{'have a '}} + \text{'very good day'}$
 $\text{'have a ' } + \underline{\text{'very good day'}}$
 $\underline{\text{'have a '}} + \underline{\text{'very good day'}}$
 $\text{'have a very good day'}$

3.2 Compound Expressions

When at least one of the operands is itself an expression (as in $2 * 5 + 1$), the expression is a compound expression. Python follows the standard mathematical order of operations, so $2 * 5 + 1$ is equivalent to $(2 * 5) + 1$. Here are some examples of compound expressions:

```

2 * 5 + 1      ⇒  11
2 + 5 - 1     ⇒   6
4 * 6 / 8     ⇒   3
True and not False ⇒  True

```

You can use parentheses to override Python's order of operations, or just for clarity. A parenthetical expression has the form:

(EXPR)

A parenthetical expression evaluates to the same value as the enclosed subexpression, *EXPR*, does. For example, (22) evaluates to the same thing 22 does, namely 22. As another example,

```
2 * (5 + 1) ⇒ 12
```

3.2.1 Rules for Evaluation

To evaluate a compound expression to a value,

1. Use order of operations to identify the main operator (the last operator that you'll apply). For example, the main operator in $2 * 5 + 1$ is $+$, so $2 * 5 + 1$ is an addition expression.
2. Identify the operands to the main operator. Then evaluate this expression (the main operator and its two operands) as you would evaluate a binary expression.

3.2.2 Examples

Below are examples of evaluating compound expressions.

$$\begin{array}{l} 1. \quad \underline{3 * 6} + 7 \\ \quad \underline{3 * \underline{6}} + 7 \\ \quad \underline{3 * 6} + 7 \\ \quad 18 + \underline{7} \\ \quad \underline{18 + 7} \\ \quad 25 \end{array}$$

This example contains some extra underlining to emphasize the following. To evaluate $3 * 6 + 7$, it is necessary to first evaluate its left-hand side, $3 * 6$. To evaluate $3 * 6$, it is necessary to first evaluate *its* left-hand side, 3.

$$\begin{array}{l} 2. \quad \underline{6} + 7 + 8 \\ \quad 6 + \underline{7} + 8 \\ \quad \underline{6 + 7} + 8 \\ \quad 13 + \underline{8} \\ \quad 13 + 8 \\ \quad 21 \end{array}$$

To simplify this document, from now on we will sometimes elide some steps if they are obvious. For instance, the following example goes straight from $6 < 0$ to `False`.

$$\begin{array}{l} 3. \quad \underline{6 < 0} \text{ or } 6 > 10 \\ \quad \text{False or } \underline{6 > 10} \\ \quad \underline{\text{False or False}} \\ \quad \text{False} \end{array}$$

3.3 Unary Expressions

A unary operator operates on a single value. In general, a unary expression has the form:

UN.OP EXPR

Two common unary operators are `not` and `-`. The `not` operator negates a Boolean value; for example, `not False` evaluates to `True`. Used as a unary operator, the `-` operator negates a numerical value; for example, `-(2 * 5)` evaluates to `-10`.

3.3.1 Rules for Evaluation

To evaluate a unary expression to a value,

1. Evaluate *EXPR* to a value and replace *EXPR* with that value.
2. Apply *UN.OP* to the value and replace the entire expression with the new value.

3.3.2 Examples

Below are examples of evaluating unary expressions.

1. $-(12 + 4)$

$$\begin{array}{r} \underline{-(16)} \\ -16 \\ -16 \end{array}$$

2. $-(1-2)$

$$\begin{array}{r} \underline{-(-1)} \\ - -1 \\ 1 \end{array}$$

3. $1 + -3$

$$\begin{array}{r} 1 + \underline{-3} \\ \underline{1 + -3} \\ -2 \end{array}$$

4. not True

$$\begin{array}{r} \underline{\text{not True}} \\ \text{False} \end{array}$$

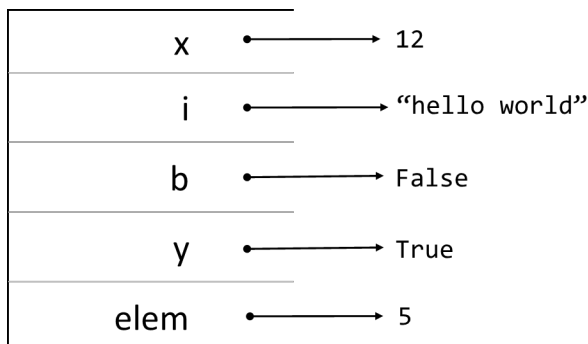
4 Variables

Think of a variable as a container. A variable stores a value so that you can reuse it later in your program. This reduces redundancy, improves performance, and makes your code more readable. In order to use a variable, you first store a value in the variable by *assigning* the variable to this value. Later, you *access* that variable, which looks up the value you assigned to it. It is an error to access a variable that has not yet been assigned. You can reassign a variable — that is, give it a new value — any number of times.

Note that Python’s concept of a variable is different from the mathematical concept of a variable. In math, a variable’s value is fixed and determined by a mathematical relation. In Python, a variable is assigned a specific value at a specific point in time, and it can be reassigned to a different value later during a program’s execution.

Python stores variables and their values in a structure called a *frame*. A frame contains a set of *bindings*. A binding is a relationship between a variable and its value. When a program assigns a variable, Python adds a binding for that variable to the frame (or updates its value if the variable already exists). When a program accesses a variable, Python uses the frame to find a binding for that variable.

Below is an illustration of a Python frame with bindings for 5 variables:



For now, this document will consider simple variable assignments and accesses. For a program with functions and function calls, Section 8 defines more complete procedures for variable assignments and accesses.

4.1 Variable Access Expressions

Variable access expressions let you use the value of a variable you've assigned. Suppose that the frame is the one illustrated above, where a variable with the name `x` is assigned the value 13. Then the expression `x` evaluates to the value 13. Here are some examples of variable access expressions:

```
answer           ⇒ 42
(answer + 2) / 2 ⇒ 22
```

In general, a variable access expression has the form:

VAR_EXPR

For now, *VAR_EXPR* is a variable name. Section 6.2 generalizes this assumption, enabling accessing of data structures, as in the expression `mylist[22]`.

4.1.1 Rules for Evaluation

To evaluate a variable access expression to a value, search the frame for a binding from *VAR_EXPR* to a value. If such a binding exists, replace the access expression with that variable's value. Otherwise raise an error, because the variable is not defined.

Later, Section 8 introduces Python functions. When accessing variables in the body of a function, use the refined rules for evaluation in Section 8.2.

4.1.2 Examples

Below are examples of evaluating variable access expressions. Each example's first line is the variable access expression, and the second is the value to which that expression evaluates. The examples are evaluated in the context of the frame presented above:

1. `x`
12
2. `i`
"hello world"
3. `o`
ERROR. The variable `o` is not defined.
4. `y`
True

4.2 Variable Assignment Statements

An assignment statement creates a variable and sets its value, or changes the value of an existing variable. Here are some examples of assignments:

```
x = 18
y = x + 1
y = y * 2
```

In general, an assignment statement has the form:

$VAR_EXPR = EXPR$

4.2.1 Rules for Evaluation

To execute an assignment statement,

1. Evaluate $EXPR$ to a value and replace $EXPR$ with that value.
2. If the variable already exists in the frame, change its binding so that it now refers to the value from the previous step. Otherwise, create a new variable in the current frame and bind it to the value from the previous step.

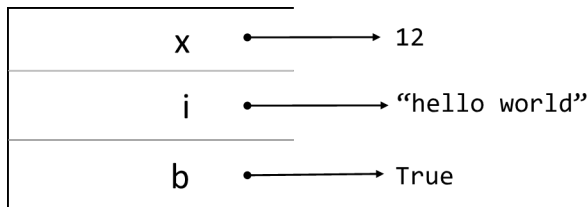
Note that an assignment statement is treated differently from expressions. For most expressions (such as $x + 18$), both of the subexpressions are evaluated to a value, then a math operation is performed. For an assignment, the right-hand side is evaluated to a value, but the left-hand side is treated as a name rather than evaluated.

The left-hand side can be more complex than simply a name. Section 6 will show how to evaluate an assignment such as `mylist[22] = EXPR`.

When assigning variables in the body of a function, use the refined rules for evaluation in Section 8.3.

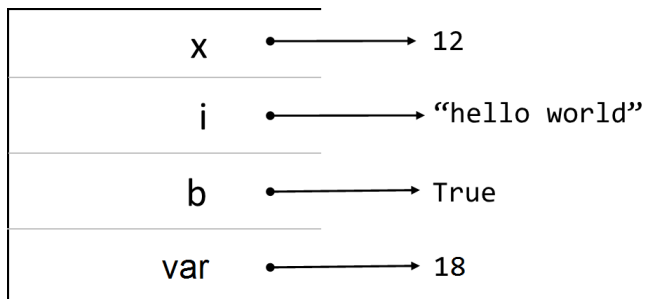
4.2.2 Examples

Below are examples of executing variable assignment statements. Each example is executed in the context of this frame, unaffected by previous examples:

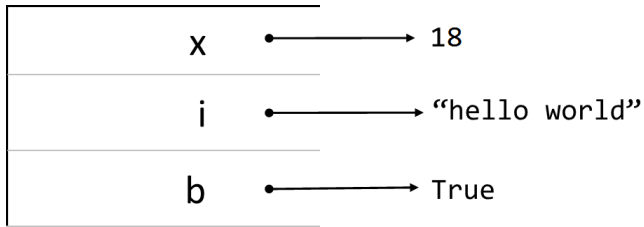


Each example shows the final frame.

1. `var = 18`
Binds new variable `var` to 18.

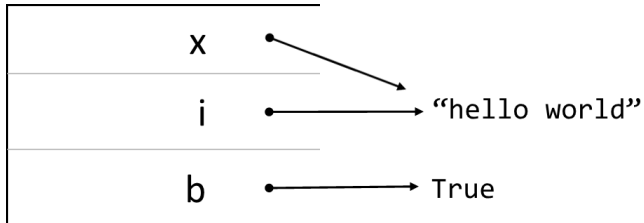


2. `x = 18`
Re-binds `x` to 18.



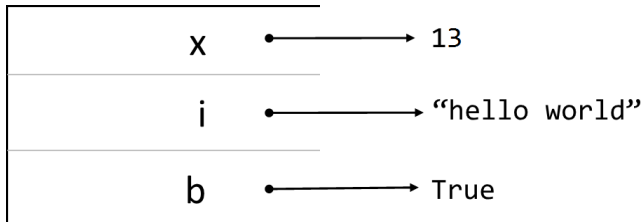
3. `x = i`

Evaluates `i` to "hello world", then re-binds `x` to that value.



4. `x = x + 1`

Evaluates `x + 1` to 13, then re-binds `x` to that value.



5 If Statements

An `if` statement lets you execute code only in certain cases (for example, only if a particular variable is less than 0). For example, here's an `if` statement that computes the absolute value of a variable `x`.

```
# x is some number
if x >= 0:
    y = x
else:
    y = -x
# y is |x|
```

An `if` statement consists of a series of `if`, `elif`, and `else` clauses. Every clause (except `else`) consists of a condition (a Boolean expression) and a body (a sequence of statements). In general, an `if` statement has the form:

```
if BOOL_EXPR:
    BODY_STATEMENTS
elif BOOL_EXPR:
    BODY_STATEMENTS
```

```

elif BOOL_EXPR:
    BODY_STATEMENTS
:
else:
    BODY_STATEMENTS

```

An `if` statement always has exactly one leading `if` clause, zero or more `elif` clauses that follow it, and zero or one `else` clause at the end.

5.1 Rules for Evaluation

In general, Python evaluates the clauses' conditions in order until one evaluates to `True`, and then executes only the statements in that clause (and not in any later clause, even if a later clause's condition is also true). To execute an `if` statement,

1. If an `else` clause exists, replace it with an `elif True` clause.
2. For each clause, from top to bottom, do the following:
 - (a) Evaluate the current clause's condition to a Boolean value (that is, `True` or `False`) and replace the condition with that value. If the condition evaluates to a non-Boolean value, convert the value to a Boolean value (see <https://docs.python.org/2/library/stdtypes.html#truth-value-testing>).
 - (b) Choose one of the following:
 - If the condition evaluates to `True`** Execute the statements inside this clause, then end execution of the `if` statement. Ignore all subsequent clauses.
 - If the condition evaluates to `False`** Ignore this clause and continue to the next clause. If there is no remaining clause, end execution of the `if` statement.

5.2 Examples

Below are examples of executing `if` statements. Each example shows an `if` statement and a timeline that shows how Python evaluates that `if` statement, and ends by showing the body statements that Python will execute. The examples are evaluated in the context of the following frame.

Each example is executed in the context of this frame, unaffected by previous examples:

<code>x</code>	→ 12
<code>j</code>	→ "hello world"
<code>b</code>	→ False
<code>y</code>	→ True
<code>elem</code>	→ 5

1. `if y:`
BODY_STATEMENTS
↓
`if True:`

BODY_STATEMENTS

The first clause is executed, because `y` evaluates to `True`.

- 2. `if b:`
 BODY_STATEMENTS
 ↓
 (Nothing)

No clause is executed, because `b` evaluates to `False`.

- 3. `if not b:`
 BODY_STATEMENTS
`else:`
 BODY_STATEMENTS
 ↓
 `if not b:`
 BODY_STATEMENTS
 `elif True:`
 BODY_STATEMENTS
 ↓
 `if True:`
 BODY_STATEMENTS
 `elif True:`
 BODY_STATEMENTS

The first clause is executed because `not b` evaluates to `True`

- 4. `if b:`
 BODY_STATEMENTS
`else:`
 BODY_STATEMENTS
 ↓
 `if b:`
 BODY_STATEMENTS
 `elif True:`
 BODY_STATEMENTS
 ↓
 `if False:`
 BODY_STATEMENTS
 `elif True:`
 BODY_STATEMENTS
 ↓
 `if False:`
 BODY_STATEMENTS
 `elif True:`
 BODY_STATEMENTS

The second clause is executed because `b` evaluates to `False`.

- 5. `if x < 0:`
 BODY_STATEMENTS
`elif x == 0:`

```

    BODY_STATEMENTS
elif x > 0:
    BODY_STATEMENTS
↓
if False:
    BODY_STATEMENTS
elif x == 0:
    BODY_STATEMENTS
elif x > 0:
    BODY_STATEMENTS
↓
if False:
    BODY_STATEMENTS
elif False:
    BODY_STATEMENTS
elif x > 0:
    BODY_STATEMENTS
↓
if False:
    BODY_STATEMENTS
elif False:
    BODY_STATEMENTS
elif True:
    BODY_STATEMENTS

```

The third clause is executed because $x < 0$ evaluates to `False`, $x == 0$ evaluates to `False`, and $x > 0$ is `True`.

```

6. if x > 0:
    BODY_STATEMENTS
    if x == 12:
        BODY_STATEMENTS
    else:
        BODY_STATEMENTS
↓
if x > 0:
    BODY_STATEMENTS
    if x == 12:
        BODY_STATEMENTS
    elif True:
        BODY_STATEMENTS
↓
if True:
    BODY_STATEMENTS
    if x == 12:
        BODY_STATEMENTS
    elif True:
        BODY_STATEMENTS

```

The first clause is executed because $x > 0$ evaluates to `True`.

6 Data structures: Lists, Tuples, Sets, and Dictionaries

So far, each value we have seen is a single datum, such as an integer, decimal number, or Boolean. Python also supports compound values, or data structures. A data structure contains multiple values. Examples include strings, lists, tuples, sets, and dictionaries.

Types of data structures

A **string** is a list of characters. It is used to represent text. Strings also have a corresponding literal constructor (such as "a string").

A **tuple** contains any number of elements (but usually only two or three). Some functions return tuples when it is convenient to return a pair of values instead of just one. Tuples are immutable, which means they cannot be changed after they're created. The values in a tuple need not be of the same type, but the values are related — for instance, a tuple might contain a string that represents a word and the number of times that word appears in a particular text file.

Here is an example of a 2-tuple whose elements are a word and the number of times that word appears in a text file: ^{tuple}

the	1112
-----	------

.

A **list** is an ordered sequence of elements. A list generally contains many elements of the same type. Lists are used when order is important (for instance, sorting) or when it is necessary to keep multiple instances of a particular element. Once you have created a list, you can change, add, and remove elements.

Here is an example of a list of words in alphabetical order: ^{list}

"fun"	"is"	"programming"	"Python"
-------	------	---------------	----------

.

Strings, tuples, and lists are all **sequences**. A sequence stores elements in a fixed order by giving each element a unique integer index in the interval $[0, n-1]$, where n is the length of the sequence. The first element has index 0, and the last has index $n-1$. You can access a particular element in a sequence by using that element's index; for example, the expression `mylist[0]` evaluates to the first element in the list `mylist`.

A **set** is an unordered collection of unique elements. Python ensures uniqueness for you — adding to the set an element that is already in the set does nothing. It is fast to determine whether a particular element is in the set, even for a large set.

Here's an example of a set that contains the names of several popular computer operating systems:

^{set}

"Windows"	"Mac OS"	"Linux"
-----------	----------	---------

.

A **dictionary** stores a set of pairs called key-value pairs. If a dictionary contains a particular key-value pair, then we say the key "maps to" the value. Given a key, a dictionary can return the value to which that key maps (not the reverse, however). A dictionary is used to associate two pieces of data (for example, the number of times a given word appears in a text file could be represented with a word as a key and the number of occurrences as a value).

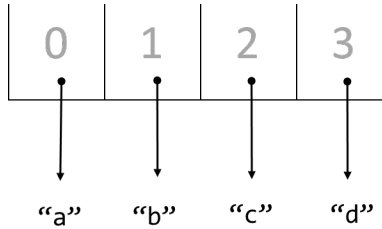
Here's an example of a dictionary that contains a set of mappings from a word to the number of times that word appears in a text file. For any given word, you can ask this dictionary the number of times the word appears: ^{dict}

"student" → 4	"programmer" → 23	"fun" → 5000	"arachnid" → 23
---------------	-------------------	--------------	-----------------

.

Visualizing data structures

A list is not so much a sequence of elements as it is a sequence of references to elements. For example, here's how you would draw a list containing the elements "a", "b", "c", and "d":



A list is similar to a frame. In a frame (list), a variable (index) is bound to a value (element), and you can retrieve the value (element) by using the variable (index). You can see that this list has four references, each of which points to one of the elements in a list. And, because this is a list, each reference has an integer index (0, 1, 2, or 3). You can use the index to access or assign a particular reference in the list.

For brevity, this document uses syntax like $\overset{\text{list}}{\boxed{\text{"a"} \ \text{"b"} \ \text{"c"} \ \text{"d"}}$ to represent the list above in text.

6.1 Constructor Expressions

A constructor expression creates a new set, list, tuple, or dictionary. Here are some constructor expressions:

- $[1, 2, 3]$ \Rightarrow A list with elements 1, 2, and 3, in that order:
- $\{1, 2\}$ \Rightarrow A set with elements 1 and 2: $\overset{\text{set}}{\boxed{1 \ 2}}$ or equivalently $\overset{\text{set}}{\boxed{2 \ 1}}$
- $\{\text{'a':1, 'b':2}\}$ \Rightarrow A dictionary in which "a" maps to 1 and "b" maps to 2: $\overset{\text{dict}}{\boxed{\text{"a"} \rightarrow 1 \ \text{"b"} \rightarrow 2}}$
or equivalently $\overset{\text{dict}}{\boxed{\text{"b"} \rightarrow 2 \ \text{"a"} \rightarrow 1}}$
- $(\text{'hello'}, 5)$ \Rightarrow A tuple with elements "hello" and 5, in that order: $\overset{\text{tuple}}{\boxed{\text{"hello"} \ 5}}$

In general, a constructor expression has the form:

List $[EXPR, EXPR, \dots, EXPR]$

Set $\{EXPR, EXPR, \dots, EXPR\}$

Tuple $(EXPR, EXPR, \dots, EXPR)$

Dictionary $\{KEY_EXPR : VAL_EXPR, \dots, KEY_EXPR : VAL_EXPR\}$

A string constructor is a string literal such as "Hello world".

6.1.1 Rules for Evaluation

To evaluate a constructor expression to a value, use these rules.

Lists, Sets, and Tuples

1. From left to right, for each expression $EXPR$ in the comma-separated list, evaluate $EXPR$ to a value and replace $EXPR$ with that value.
2. Replace the constructor expression with a list, set, or tuple value containing exactly the values from the previous step. Retain order for a list or tuple. Remove duplicate elements for a set.

Dictionaries

1. For each expression *KEY_EXPR* : *VAL_EXPR* in the comma-separated list, from left to right, do the following:
 - (a) Evaluate *VAL_EXPR* to a value and replace *VAL_EXPR* with that value.
 - (b) Evaluate *KEY_EXPR* to a value and replace *KEY_EXPR* with that value.
2. Replace the constructor expression with a dictionary containing exactly the mappings from the previous step. If there are multiple mappings for a particular key, use the last (rightmost) mapping in the constructor expression.

6.1.2 Examples

Below are examples of evaluating constructor expressions. Each example contains a constructor and a description of the data structure the constructor creates.

1. `[0, 1, 0]`

Creates a new list that contains the values 0, 1, and 0 at the indices 0, 1, and 2, respectively:

0	1	0
---	---	---

2. `[]`

Creates a new, empty list:

--

3. `(3.14159, 'pi')`

Creates a new two-tuple with the values 3.14159 and “pi” at the indices 0 and 1, respectively:

3.14159	“pi”
---------	------

4. `(12,)`

Creates a new 1-tuple with the value 12 at index 0:

12

. Note the trailing comma, which makes Python interpret the value inside the parentheses as a tuple rather than a parenthetical expression (as discussed in Section 3.2). A 1-tuple isn’t very useful (you may as well just use a single value), so you should avoid using them.

5. `()`

Creates a new, empty tuple:

--

. It’s rarely useful to create a tuple with no elements.

6. `{'a':1, 'b':2, 'c':3}`

Creates a new dictionary in which the key “a” maps to the value 1, “b” maps to 2, and “c” maps to

3:

“a”→2	“b”→2	“c”→3
-------	-------	-------

7. `{'a':1, 'a':0, 'b':2, 'a':3}`

Creates a new dictionary in which the key “b” maps to the value 2 and “a” maps to 3. Note that the

rightmost mapping for “a” overwrites all previous mappings for that key:

“b”→2	“a”→3
-------	-------


8. `{'a':1, 'b':1}`

Creates a new dictionary in which “a” and “b” both map to 1. Note that although keys in a dictionary

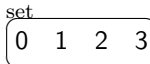
must be unique, the values need not be:

“a”→1	“b”→1
-------	-------

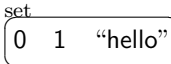
9. {}

Creates a new, empty dictionary: 


10. {0, 1, 2, 3}

Creates a new set that contains the values 0, 1, 2, and 3: 

11. {0, 1, 1, 'hello'}


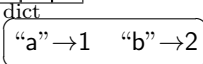
Creates a new set that contains the values 0, 1, and "hello": 

12. set()

Creates a new, empty set: . The expression {} creates an empty dictionary, not an empty set.

6.2 Data Structure Access Expressions

Here are some examples of access expressions (accessing an element of a data structure), the following variables are defined:

```
# lst is 
# dict is 
```

```
lst[0]    => 1
lst[1]    => 2
lst[-1]   => 3
lst[-2]   => 2
dict['a'] => 1
```

Sequences and dictionaries all provide methods for retrieving specific values from the structure. A dictionary access takes a key; the dictionary access expression evaluates to the value associated with that key. A sequence access takes an index; the sequence access expression evaluates to the element at that index. The following sections describe how to evaluate access expressions. An access expression has the general form:

EXPR [*INDEX_EXPR*]

Note that sets don't provide a way to access specific elements. If, in your code, you need to access a particular element in a set, then consider using a different data structure.

6.2.1 Rules for Evaluation

To evaluate *EXPR* [*INDEX_EXPR*] to a value

This expression returns a single value in the data structure given by *EXPR*.

1. Evaluate *EXPR* to a value and replace *EXPR* with that value.
2. Evaluate *INDEX_EXPR* to a value and replace *INDEX_EXPR* with that value.
3. If *EXPR* evaluates to something other than a sequence or dictionary, then raise an error.

4. **If *EXPR* is a sequence** If *INDEX_EXPR* is not an integer on the interval $[-n, n - 1]$ (inclusive, where n is the length of the sequence), then this access fails.

If *INDEX_EXPR* is negative, replace *INDEX_EXPR* with $n - |\text{INDEX_EXPR}|$. (Note that accessing index -1 is equivalent to accessing the last element in the sequence.)

Replace the access expression with the value in the sequence at that index. Sequences are zero-indexed, so the first value in the sequence is at index 0.

If *EXPR* is a dictionary The expression succeeds only if *INDEX_EXPR* is a key in the dictionary. Replace the entire access expression with the value to which the key maps.

6.2.2 Examples

Below are examples of evaluating access expressions. Each example contains an access expression and the value to which that access evaluates.

Each example is executed in the context of this frame, unaffected by previous examples:

- mylist is

list			
0	5	3	4

- mydict is

dict		
"a" → 1	"b" → 2	"c" → "a"

- mytuple is

tuple	
"hello"	5

1. mylist[0]

list			
0	5	3	4

 [0]

list			
0	5	3	4

 [0]

0

2. mylist[4]

ERROR. No index 4.

3. mylist[3]

4

4. mylist[-1]

list			
0	5	3	4

 [-1]

list			
0	5	3	4

 [len(mylist)-1]

list			
0	5	3	4

 [4-1]

list			
0	5	3	4

 [3]

4

5. mylist[0 - 3]

list			
0	5	3	4

 [0 - 3]

list			
0	5	3	4

 [-3]

```
list
0 5 3 4 [len(mylist)-3]
list
0 5 3 4 [4-3]
list
0 5 3 4 [1]
5
```

6. `mydict['a']`

```
dict
"a" → 1  "b" → 2  "c" → "a" ['a']
dict
"a" → 1  "b" → 2  "c" → "a" ["a"]
1
```

7. `mydict[mydict['c']]`

```
mydict[mydict["c"]]
mydict["a"]
1
```

We skipped evaluating `mydict` to `dict` `"a" → 1 "b" → 2 "c" → "a"` in this example, for brevity.

8. `mydict[(12, 13)]`

```
ERROR. The value tuple
12 13 is not a key in the dictionary.
```

9. `mytuple[0]`

```
tuple
"hello" 5 [0]
tuple
"hello" 5 [0]
"hello"
```

10. `mytuple[1]`

```
5
```

11. `mytuple[-1]`

```
tuple
"hello" 5 [-1]
tuple
"hello" 5 [-1]
tuple
"hello" 5 [len(mytuple)-1]
tuple
"hello" 5 [2-1]
mytuple[1]
5
```

6.3 Data Structure Assignment Statements

In the same way that sequences and dictionaries provide a way to access individual elements, they also provide a way to reassign individual elements. This lets you change the elements of the structure. Here are some examples of reassigning an element of a data structure:

```
# lst is 

|      |   |   |
|------|---|---|
| list |   |   |
| 1    | 2 | 3 |


# dict is 

|         |         |
|---------|---------|
| dict    |         |
| "a" → 1 | "b" → 2 |


```

```
lst[0] = 17      after this, lst is 

|      |   |   |
|------|---|---|
| list |   |   |
| 17   | 2 | 3 |


lst[-1] = 18    after this, lst is 

|      |   |    |
|------|---|----|
| list |   |    |
| 1    | 2 | 18 |


dict['b'] = 12  after this, dict is 

|         |          |
|---------|----------|
| dict    |          |
| "a" → 1 | "b" → 12 |


```

In general, an element assignment statement has the form:

DS_EXPR [*EXPR*] = *ASSIGN_EXPR*

Element assignments are very similar to variable assignments. There is an l-expression (on the left of the assignment operator; in the example above, it is *DS_EXPR* [*EXPR*]) and an r-expression (on the right; in the example, it is *ASSIGN_EXPR*). The l-expression is not evaluated to a value; rather, it identifies a particular reference or binding in the structure. The assignment makes that reference point to the value to which r-expression evaluates.

Tuples and sets don't provide a way to reassign specific values. It isn't surprising that sets don't support this operation, because sets don't even provide a way to access specific values. Tuples, on the other hand, do not support this operation because they are immutable: once created, they cannot change.

6.3.1 Rules for Evaluation

Assigning a single element makes that particular reference point to a different value. To execute an element assignment statement,

1. Evaluate the r-expression to a value and replace the r-expression with that value.
2. Evaluate the l-expression to identify a particular reference in the data structure. When accessing a single index in a list, this statement will fail if the index is not in the list. However, when accessing a key in a dictionary, if the key is not in the dictionary already then Python will add it and the l-expression will identify a reference for this new key.
3. Change the reference so that it points to the value given by the r-expression.

6.3.2 Examples

Below are examples of executing assignment statements. Each example contains an assignment statement, some intermediate steps of executing the statement shown underneath, and the final value of the affected data structure.

Each example is executed in the context of this frame, unaffected by previous examples:

- lst is

list		
1	2	3
- dict is

dict	
"a" → 1	"b" → 2

- t is

tuple
"hello" 5

1. $lst[0] = 17$
 $lst[0] = 17$

list
1 2 3

 $[0] = 17$

list
1 2 3

 $[0] = 17$

Now, $lst \Rightarrow$

list
17 2 3

.

Note that in this example we did not evaluate the l-value $lst[0]$ to a value; rather, we evaluated the parts and then used them to assign one of the slots of lst .

2. $dict[lst[0]] = 'c'$
 $dict[lst[0]] = "c"$

dict
"a" → 1 "b" → 2

 $[lst[0]] = "c"$

dict
"a" → 1 "b" → 2

 $[1] = "c"$

Now, $dict \Rightarrow$

dict
"a" → 2 "b" → 2 1 → "c"

.

3. $dict['a'] = lst[1]$
 \vdots

dict
"a" → 1 "b" → 2

 $["a"] = 2$

Now, $dict \Rightarrow$

dict
"a" → 2 "b" → 2

.

4. $t[1] = 6$
 ERROR: 'tuple' object does not support item assignment.
 (That is Python's way of saying that t 's value is immutable.)

5. $dict['a'] = lst$

dict
"a" → 1 "b" → 2

 $[dict["a"]] = lst$
 Now, $dict \Rightarrow$

dict		
"a" → <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="text-align: center; padding: 2px;">list</td></tr><tr><td style="padding: 2px;">1 2 3</td></tr></table> "b" → 2	list	1 2 3
list		
1 2 3		

 where

list
1 2 3

 is the same list value as lst has

6.4 Sequence Slice Access and Assignment Expressions

Just as a list access evaluates to a single element of the list, a slice operation evaluates to multiple elements of the list — that is, a subsequence. You can also assign to a slice, which modifies the original sequence by adding or replacing a sequence of elements. Here are some examples of slice accesses and assignments.

lst is

list
1 2 3

$lst[0:2] \Rightarrow$

list
1 2

$lst[0:2] = ['a', 'b', 'c']$ after this, lst is

list
"a" "b" "c" 3

In general, a list slice access expression has the form:

EXPR [*INDEX_EXPR_1* : *INDEX_EXPR_2*]

In general, a list slice assignment expression has the form:

DS_EXPR [*INDEX_EXPR_1* : *INDEX_EXPR_2*] = *ASSIGN_EXPR*

The rules for evaluating a slice operation have many special cases, which makes debugging slices very difficult. For this reason, you should avoid using slices for anything but the most predictable operations.

6.4.1 Rules for Evaluation

To evaluate *EXPR* [*INDEX_EXPR_1* : *INDEX_EXPR_2*] **to a value**

1. Evaluate *EXPR* to a value and replace *EXPR* with that value.
Henceforth in these rules, mentions of *EXPR* mean the value, not the original expression.
2. If the expression has any of the following forms, transform it as described below. These Python shorthands make it easy for a programmer to select the first or last few elements of the sequence.
EXPR [: *INDEX_EXPR_2*] becomes *EXPR* [0 : *INDEX_EXPR_2*]
EXPR [*INDEX_EXPR_1* :] becomes *EXPR* [*INDEX_EXPR_1* : len(*EXPR*)]
EXPR [:] becomes *EXPR* [0 : len(*EXPR*)]
3. Evaluate *INDEX_EXPR_1* to a value and replace the expression with that value. Then do the same for *INDEX_EXPR_2*.
4. Let *n* be the length of *EXPR*. If either index is less than $-n$, replace it with $-n$. If either index is greater than *n*, replace it with *n*.
5. Let *n* be the length of *EXPR*. If *INDEX_EXPR_1* is negative, replace it with $n - |\text{INDEX_EXPR_1}|$. Then do the same for *INDEX_EXPR_2*. At this point, both indices should be on the interval $[0, n]$ (note that *n* is a valid index in a slice, but not in an access to a single element).
6. Raise an error if *EXPR* is not a sequence.
7. Let *i* be the first index value, and let *j* be the second. Create a list of indices containing every integer *x* such that $i \leq x < j$. (For example, if *i* is 2 and *j* is 4, then this list will contain 2 and 3.) Note that it is possible for this list to be empty (for instance, if $j \leq i$).
8. For each *x* in this list of indices, replace *x* with the value in the original sequence at index *x*. Replace the slice access with this new list. It is important to note that the slice returns a new list, so reassigning elements of the slice does not affect the original list.

Reassigning a slice first removes that slice, then inserts a new sequence of references starting at the lowest index of the slice.

To execute *DS_EXPR* [*INDEX_EXPR_1* : *INDEX_EXPR_2*] = *ASSIGN_EXPR*

1. Evaluate the r-expression *ASSIGN_EXPR* to a value and replace *ASSIGN_EXPR* by that value.
2. Evaluate the l-expression to identify a sequence of l-references in the data structure.
3. Raise an error if (a) *DS_EXPR* is not a sequence, or (b) the r-expression is not another data structure.
4. Remove the specified slice from the original sequence (if the slice is empty, don't remove anything).

5. Create a new list containing the elements of *ASSIGN_EXPR* in iterator order.
6. Insert, starting at the lowest index in the slice, the sequence of references in the list from the previous step.

6.4.2 Examples

In these examples, `mylist` is

0	5	3	4
---	---	---	---

.

1. `mylist[0:3]`
indices 0, 1, and 2

0	5	3
---	---	---

2. `mylist[-3:0]`
`mylist[1:0]`

no indices

--

3. `mylist[-100:100]`
`mylist[-4:4]`
`mylist[0:4]`
indices 0, 1, 2, and 3

0	5	3	4
---	---	---	---

4. `mylist[1:]`
`mylist[1:len(mylist)]`
`mylist[1:4]`
indices 1, 2, and 3

5	3	4
---	---	---

5. `mylist[-3:]`
`mylist[-3:len(mylist)]`
`mylist[-3:4]`
`mylist[1:4]`
indices 1, 2, and 3

5	3	4
---	---	---

6. `mylist[:]`
`mylist[0:len(mylist)]`
`mylist[0:4]`
indices 0, 1, 2, and 3

0	5	4	3
---	---	---	---

7. `mylist[0:3] = ['hello', 'there']`
indices to replace: 0, 1, and 2

list to insert:

"hello"	"there"
---------	---------

"hello"	"there"	3
---------	---------	---

8. `mylist[0:3] = {'hello', 'there'}`
 indices to replace: 0, 1, and 2

list to insert:

list	
“hello”	“there”

 or

list	
“there”	“hello”

; Python will choose one of these arbitrarily because the argument is a set, which does not have a defined order.

list		
“hello”	“there”	3

 or

list		
“there”	“hello”	3

9. `mylist[:1] = ['a', 'b']`
`mylist[0:1] = ['a', 'b']`
 indices to replace: 0

list to insert:

list	
“a”	“b”

list				
“a”	“b”	5	3	4

10. `mylist[:] = ['b', 'c']`
`mylist[0:len(mylist)] = ['b', 'c']`
`mylist[0:4] = ['b', 'c']`
 indices to replace: 0, 1, 2, and 3

list to insert:

list	
“b”	“c”

list	
“b”	“c”

6.5 del Statements

The `del` keyword removes elements from lists and dictionaries. Here are some example statements that use the `del` keyword:

```
# lst = [1, 2, 3, 4]
# d = 'a':1, 'b':2
del lst[0]           # lst is [2, 3, 4]
del lst[0:2]        # lst is [3, 4]
del d['a']          # d is {'b':2}
```

In general, a `del` statement has the form:

```
del ACCESS_EXPR
del ACCESS_EXPR, ACCESS_EXPR
```

6.5.1 Rules for Evaluation

To execute `del ACCESS_EXPR`

1. Perform a data structure access on `ACCESS_EXPR`. The `del` statement fails if the access is not to a list or dictionary.
2. Choose one of the following:

If `DS_EXPR` is a list The access refers to particular element (or slice) of the list. Remove this element (slice) from the list.

If *DS_EXPR* is a dictionary The access refers to a key-value mapping in the dictionary. Remove this binding from the dictionary.

To execute `del ACCESS_EXPR, ACCESS_EXPR, ...`

This statement lets you delete several items in a single line of code. Convert it into a sequence of single-element `del` statements.

6.5.2 Examples

Below are examples of executing `del` statements. Each example contains a `del` statement and the final value of the affected data structure(s). In the following examples, the following variables are defined.

Each example starts with these variables defined, unaffected by previous examples.

• `lst4` is

0	1	2	3
---	---	---	---

• `dict` is

"a" → 1	"b" → 2
---------	---------

1. `del lst4[0]`

1	2	3
---	---	---

2. `del lst4[-1]`

0	1	2
---	---	---

3. `del lst4[:]`

--

4. `del lst4[0:2]`

2	3
---	---

5. `del dict['a']`

"b" → 2

6. `del dict['c']`

ERROR. `KeyError ('c' is not a key in dict)`

7. `del dict['a'], dict['c']`

First deletes `dict['a']`, then throws an error on `dict['c']`. At the end, `dict` is

"b" → 2

8. `del lst4[0], dict['b']`

`lst4` is

1	2	3
---	---	---

 and `dict` is

"a" → 1

7 Loop Statements

7.1 for Loop Statements

A `for` loop executes its body multiple times. The `for` loop *iterates* over a sequence such as a list or string, and executes the body once for each element in the sequence. A `for` loop also defines a loop variable. On each iteration, the loop variable is assigned to the next sequence element. Here is an example of a `for` loop:


```
for value in [1, 2, 6]:
    print value + 1
```

This code prints:

```
2
3
7
```

In general, a `for` loop has the form:

```
for LOOP_VAR in DS_EXPR:
    BODY_STATEMENTS
```

7.1.1 Rules for Evaluation

We give two rules for evaluating `for` loop: a control-oriented version, and a rewriting version. They produce identical results, and you can use whichever one you prefer.

To execute a `for` loop,

1. Evaluate *DS_EXPR* to a value and replace the expression with that value.
2. Assign the *LOOP_VAR* to the first value in the sequence. If the sequence is empty (contains no elements), exit the loop.
3. Execute the statements in the body of the loop. If you encounter a `break` statement, immediately exit the loop. If you encounter a `continue` statement, immediately continue to the next step. For more information, see Section 7.3 on the `break` statement and Section 7.4 on the `continue` statement.
4. Reassign *LOOP_VAR* to the next value in the sequence, then repeat the previous step. If there is no next element, exit the loop. For lists, the next element is the element at the next index. For sets and dictionaries, the order is undefined, so the next element can be any of the remaining elements.

You can also evaluate a loop by rewriting it. Rewriting a loop is too cumbersome to use regularly, but it's unambiguous and therefore can yield insight. To rewrite a `for` loop,

1. Evaluate the *DS_EXPR* to a value.
2. Write an assignment to the loop variable for each element in *DS_EXPR* (if the data structure is a sequence such as a list, make sure the assignments are in the same order as in the list). For example, if the sequence had four elements, you would write four assignments.
3. Write a copy of the loop's body after each assignment.
4. Execute the resultant statements.

If you encounter a `break` statement, immediately skip to the end of the loop statements by deleting them all — that is, rewriting them to nothing.

If you encounter a `continue` statement, immediately skip to the next assignment of the loop variable *LOOP_VAR* by deleting all statements up to that point — that is, rewriting them to nothing.

For more information, see Section 7.3 on the `break` statement and Section 7.4 on the `continue` statement.

7.1.2 Examples

Below are examples of executing `for` loops, using the rewriting approach. Each example contains a `for` loop and a sequence of assignments to the loop variable in the correct order. (For sets and dictionaries, remember that the order is undefined.)

1. (a) `for y in ['hello', 'there', 'hi']:`
 `BODY_STATEMENTS`
(b) `y = 'hello'`
 `BODY_STATEMENTS`
 `y = 'there'`
 `BODY_STATEMENTS`
 `y = 'hi'`
 `BODY_STATEMENTS`
2. (a) `for x in [17, 12, 14, 13, 14, 18]:`
 `BODY_STATEMENTS`
(b) `x = 17`
 `BODY_STATEMENTS`
 `x = 12`
 `BODY_STATEMENTS`
 `x = 14`
 `BODY_STATEMENTS`
 `x = 13`
 `BODY_STATEMENTS`
 `x = 14`
 `BODY_STATEMENTS`
 `x = 18`
 `BODY_STATEMENTS`
3. (a) `for x in [1, 3]:`
 `for y in [2, 4]:`
 `BODY_STATEMENTS`
(b) `x = 1`
 `y = 2`
 `BODY_STATEMENTS`
 `y = 4`
 `BODY_STATEMENTS`
 `x = 3`
 `y = 2`
 `BODY_STATEMENTS`
 `y = 4`
 `BODY_STATEMENTS`
4. (a) `for x in [1, 1, 3]:`
 `BODY_STATEMENTS_1`
 `for y in {'a', 'b', 'c'}:`
 `BODY_STATEMENTS_2`
(b) `x = 1`
 `BODY_STATEMENTS_1`
 `y = 'a'`
 `BODY_STATEMENTS_2`

```

y = 'b'
BODY_STATEMENTS_2
y = 'c'
BODY_STATEMENTS_2
x = 1
BODY_STATEMENTS_1
y = 'a'
BODY_STATEMENTS_2
y = 'b'
BODY_STATEMENTS_2
y = 'c'
BODY_STATEMENTS_2
x = 3
BODY_STATEMENTS_1
y = 'a'
BODY_STATEMENTS_2
y = 'b'
BODY_STATEMENTS_2
y = 'c'
BODY_STATEMENTS_2

```

5. (a) for x in [15, 10, 12, 13, 17, 14, 16]:

```

    if x >= 15:
        for y in ['a', 'b']:
            BODY_STATEMENTS_1
        BODY_STATEMENTS_2

```

(b) x = 15

```

    if x >= 15:
        for y in ['a', 'b']:
            BODY_STATEMENTS_1
        BODY_STATEMENTS_2
    x = 10
    if x >= 15:
        for y in ['a', 'b']:
            BODY_STATEMENTS_1
        BODY_STATEMENTS_2
    x = 12
    if x >= 15:
        for y in ['a', 'b']:
            BODY_STATEMENTS_1
        BODY_STATEMENTS_2
    x = 13
    if x >= 15:
        for y in ['a', 'b']:
            BODY_STATEMENTS_1
        BODY_STATEMENTS_2
    x = 17
    if x >= 15:
        for y in ['a', 'b']:
            BODY_STATEMENTS_1
        BODY_STATEMENTS_2
    x = 14

```

```

if x >= 15:
    for y in ['a', 'b']:
        BODY_STATEMENTS_1
BODY_STATEMENTS_2
x = 16
if x >= 15:
    for y in ['a', 'b']:
        BODY_STATEMENTS_1
BODY_STATEMENTS_2

```

6. (a) `for x in {'you', 'will', 'consume', 'it'}:`
 `BODY_STATEMENTS`
- (b) `x = 'consume'`
 `BODY_STATEMENTS`
 `x = 'you'`
 `BODY_STATEMENTS`
 `x = 'it'`
 `BODY_STATEMENTS`
 `x = 'will'`
 `BODY_STATEMENTS`

7.2 while Loop Statements

A `while` loop is like a `for` loop in that it executes a block of statements repeatedly. However, whereas a `for` loop executes exactly one time for each element in a data structure, a `while` loop executes an indefinite number of times while a condition is `True` (or, conversely, until a condition is `False`). Here's an example of a `while` loop that computes x to the power y , or x^y :

```

# Computes x**y
result = 1
while y > 0:
    result = result * x
    y = y - 1
# result = x**y

```

A `while` loop has the general form below. Every `while` loop includes a Boolean condition and some statements in its body.

```

while BOOL_EXPR:
    BODY_STATEMENTS

```

The `while` loop can be confusing at first. It is easy to think that the loop exits the instant the condition becomes false, even if that happens in the middle of the loop, but this is not true. The condition is only checked once per iteration, at the beginning of that iteration, and if the condition is true then the entire body executes before the condition is checked again.

To prevent this confusion, you should think of a `while` loop as an `if` statement that repeats over and over again. That is, if the condition is true, then execute the body. If the condition is still true, execute the body, and so on until the condition is not true.

7.2.1 Rules for Evaluation

To execute a `while` loop,

1. Evaluate the condition of the loop. If the condition evaluates to a non-Boolean value, convert the value to a Boolean value (see <https://docs.python.org/2/library/stdtypes.html#truth-value-testing>). If the value is `False`, exit the loop.
2. Execute the entire body of the loop. After executing the entire body, go to the previous step. If you encounter a `break` statement, immediately exit the loop. If you encounter a `continue` statement, immediately continue to the next step. For more information, see Section 7.3 on the `break` statement and Section 7.4 on the `continue` statement.

(It is possible to give a rewriting semantics for loops, similarly to how Section 7.1.1 gives both a control-oriented and a rewriting version of `for` loop semantics. We omit the rewriting semantics for `while` loops, for brevity.)

7.2.2 Examples

Below are examples of executing `while` loops. Each example contains a `while` loop, the values of several variables before the loop, and each variable's value after each iteration of the loop.

1. (a)

```
# lst = [1, 3, 2], i = 0
while i < len(lst):
    val = lst[i]
    i += 1
```

(b) After iteration 1:
`lst = [1, 3, 2], i = 1, val = 1`
 After iteration 2:
`lst = [1, 3, 2], i = 2, val = 3`
 After iteration 3:
`lst = [1, 3, 2], i = 3, val = 2`
2. (a)

```
# result = 1, x = 2, y = 3
while y > 0:
    result = result * x
    y = y - 1
```

(b) After iteration 1:
`result = 2, x = 2, y = 2`
 After iteration 2:
`result = 4, x = 2, y = 1`
 After iteration 3:
`result = 8, x = 2, y = 0`

(c) Now, `result` is 2^3 . The original values of `x` and `y` were 2 and 3.
3. (a)

```
# fact = 1, i = 1
while i <= 4:
    i += 1
    fact *= i
```

(b) After iteration 1:
`fact = 2, i = 2`
 After iteration 2:
`fact = 6, i = 3`
 After iteration 3:
`fact = 24, i = 4` After iteration 4:
`fact = 120, i = 5`

(c) After each iteration, `fact` is $i!$, the factorial of `i`.

7.3 break Statements

The `break` statement only appears inside the body of a `for` or `while` loop. Executing a `break` statement immediately ends execution of the loop inside which it's declared. Here's an example of a `break` statement:

```
# s is a string
i = 0
for char in s:
    if char == ' ':
        break
    i += 1
# i is the index of the first space in s
```

The `break` keyword always appears on its own line, with no other expressions on the line.

7.3.1 Rules for Evaluation

To execute a `break` statement, immediately end execution of the loop. Do not execute any more lines in the body of the loop.

If the `break` statement is enclosed in a nested loop, only end execution of the innermost loop that contains the `break` statement.

7.3.2 Examples

Below are examples of executing `break` statements. Each example contains a loop that contains a `break` statement, and is formatted in the same style as in previous sections.

- (a) `# i = 0, s = 'hello there'`

```
for char in s:
    if char == ' ':
        break
    i += 1
```

- (b) `char = 'h'`
`i = 1`
`char = 'e'`
`i = 2`
`char = 'l'`
`i = 3`
`char = 'l'`
`i = 4`
`char = 'o'`
`i = 5`
`char = ' '`

- (c) Note that `i` is the index of the first space in `s`.

- (a) `for x in [1, 2]:`
`for y in ['a', 'b', 'c']:`
`if y == 'b':`
`break`

- (b) `x = 1`
`y = 'a'`
`y = 'b'`
`x = 2`

```
y = 'a'
y = 'b'
```

7.4 continue Statements

The `continue` statement only appears inside the body of a `for` or `while` loop. Executing a `continue` statement immediately ends execution of the current iteration of the loop and skips to the next iteration (as opposed to a `break` statement, which exits the loop completely). Here is an example of a `continue` statement:

```
# lst is a list
other = []
for val in lst:
    if val is None:
        continue
    other.append(val)
# other contains all the values in lst that are not None
```

Like the `break` keyword, the `continue` keyword always appears on its own line, with no other expressions on the line. Note that in general, it is bad style to use `continue` because it is difficult to reason about.

7.4.1 Rules for Evaluation

To execute a `continue` statement, immediately move execution to the next iteration of the loop. Do not execute any more statements in the body of the loop for this iteration.

For a `while` loop, make sure to test the condition before starting the next iteration, and for a `for` loop, remember to reassign the loop variable.

Like the `break` statement, if the `continue` statement is enclosed in a nested loop, only skip to the next iteration of the innermost loop that contains the `continue` statement.

7.4.2 Examples

Below are examples of executing `continue` statements. Each example contains a loop that contains a `continue` statement, and is formatted in the same style as in previous sections.

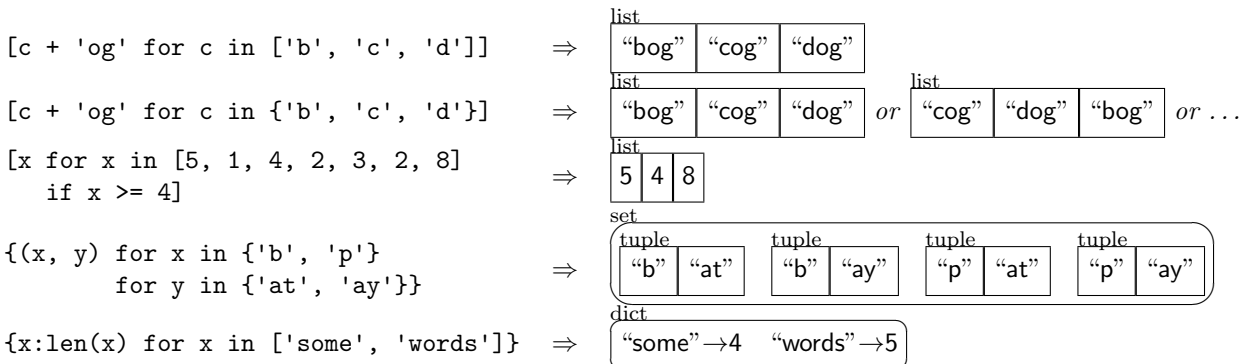
```
# Loop
# lst = [1, None, None, 4], i = 0
for val in lst:
    if val is None:
        continue
    i += 1

# Assignments
val = 1
i = 1
val = None
val = None
val = 4
i = 2

# Note that i is the number of values in lst that are not None
```

7.5 Comprehension Expressions

A comprehension provides a powerful and succinct way to generate a data structure. A comprehension can do in a single line what would otherwise take a series of nested loops and `if` statements to do. Here are some examples of comprehensions:



A comprehension expression has the general form:

List [*ELEM_EXPR GEN_CLAUSE GEN_AND_IF_CLAUSES*]

Set { *ELEM_EXPR GEN_CLAUSE GEN_AND_IF_CLAUSES* }

Dictionary { *KEY_EXPR:VAL_EXPR GEN_CLAUSE GEN_AND_IF_CLAUSES* }

Every comprehension has exactly one leading *ELEM_EXPR* (element expression), one or more *GEN_CLAUSE* (generator clause), and zero or more trailing *IF_CLAUSES*, which may be interspersed with any generator clauses after the first.

7.5.1 Rules for Evaluation

Evaluating a comprehension requires expanding the comprehension into several lines of code. The first line of code initializes a new variable to an empty data structure, and the following lines add elements to the data structure in a loop. To evaluate a comprehension expression to a value,

1. Pick a variable name that is not defined (we'll choose `struct`), then write on the line preceding the comprehension one of the following:

For a list comprehension `struct = []`.

For a set comprehension `struct = set()`.

For a dictionary comprehension `struct = {}`.

2. Identify the one or more generator and `if` clauses in the comprehension. Starting with the leftmost, and on the line following the assignment to `struct`, write these clauses as a series of progressively more nested statements. (That is, each new statement should be in the body of the previous statement. The example comprehension above should have three levels — an `if` statement inside two `for` loops.)
3. Choose one of the following:

If `struct` is a list Write the statement `struct.append(ELEM_EXPR)` inside the most nested statement.

If `struct` is a set Write the statement `struct.add(ELEM_EXPR)` inside the most nested statement.

If struct is a dictionary Write the statement `struct[KEY_EXPR] = VAL_EXPR` inside the most nested statement.

4. Replace the comprehension with a variable access to `struct`.
5. Execute the resultant statements, starting with the assignment to `struct`.

7.5.2 Examples

Below are examples of evaluating comprehension expressions. Each example contains a comprehension, the equivalent `for` and `if` statements, and the final value to which the comprehension evaluates.

1. `[len(y) for y in ['hello', 'there', 'hi']]`
 - (a)

```
struct = []
for y in ['hello', 'there', 'hi']:
    struct.append(len(y))
struct
```
 - (b) `struct` \Rightarrow

list
5 5 2
2. `[x for x in [17, 12, 14, 13, 14, 18] if x <= 13]`
 - (a)

```
struct = []
for x in [17, 12, 14, 13, 14, 18]:
    if x <= 13:
        struct.append(x)
struct
```
 - (b) `struct` \Rightarrow

list
12 13
3. `len([x for x in [17, 12, 14, 13, 14, 18] if x <= 13])`
 - (a)

```
struct = []
for x in [17, 12, 14, 13, 14, 18]:
    if x <= 13:
        struct.append(x)
len(struct)
```
 - (b) `len(struct)` \Rightarrow 2
4. (a) `[(x, y) for x in [1, 3] for y in [2, 4] if x > y]`

```
struct = []
for x in [1, 3]:
    for y in [2, 4]:
        if x > y:
            struct.append((x, y))
struct
```

 - (b) `struct` \Rightarrow

list
(3, 2)
5. `{x for x in [1, 1, 3]}`
 - (a)

```
struct = set()
for x in [1, 1, 3]:
    struct.add(x)
struct
```

(b) struct \Rightarrow

set
1 3

6. {word:len(word) for word in ['hello', 'there', 'hi']}

(a)

```
struct = {}
for word in ['hello', 'there', 'hi']:
    struct[word] = len(word)
struct
```

(b) struct \Rightarrow

dict
"hello"→5 "there"→5 "hi"→2

7. {(x, y, z) for (x, y) in [('hello', 5), ('this', 4)] for z in [10, 15]}

(a)

```
struct = set()
for (x, y) in [('hello', 5), ('this', 4)]:
    for z in [10, 15]:
        struct.add((x, y, z))
struct
```

(b) struct \Rightarrow

set
tuple
"hello" 5 10

tuple
"hello" 5 15

tuple
"this" 4 10

tuple
"this" 4 15

8 Functions

A function lets you name and reuse a particular sequence of statements. This makes your code shorter, more readable, and easier to understand and debug. Additionally, if you ever work on a program in a team, you will find that properly decomposing your program into functions makes it easy to divide the work among members of the team.

8.1 Function Definition Statements

A function definition statement creates a new function that you can use in your code. Here is an example function definition that creates a function that returns the absolute value of its parameter:

```
def absVal(x):
    if x < 0:
        x = -x
    return x
```

In general, a function definition has the form:

```
def FUNCTION_NAME(PARAM1, PARAM2, ..., PARAM_N):
    BODY_STATEMENTS
```

A function definition is similar to a variable definition: it creates a binding in Python's current frame from the the function name (here, *FUNCTION_NAME*) to a function value.

In Python, functions are values just like numbers or strings. Specifically, if you have a variable called *x* that has the value 17, and then you define a function called *x*, Python will reassign the existing *x* to have a function value instead of an integer value. To avoid this often-confusing mistake in your own code, avoid creating functions and variables with the same name.

8.1.1 Rules for Evaluation

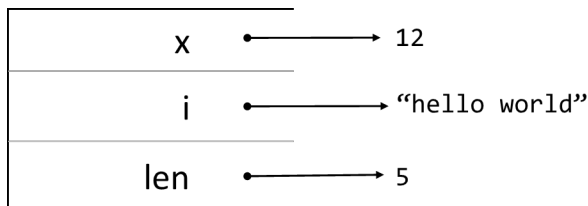
To execute a function definition statement,

1. Create a new function value. This function has formal parameters *PARAM1* through *PARAM_N*. When called, it will execute the statements *BODY_STATEMENTS*.
2. Perform a variable assignment (see section 4.1) with the symbol *FUNCTION_NAME*. Give this variable the function value you created in the previous step.

8.1.2 Examples

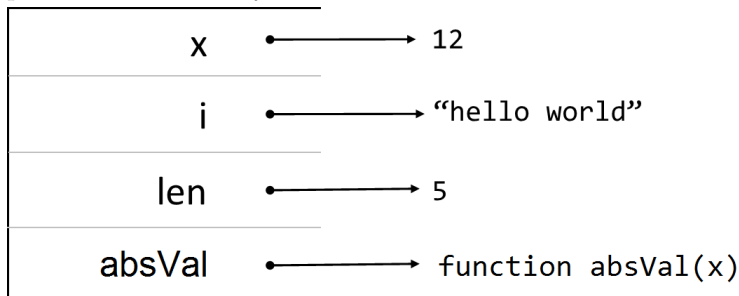
Below are examples of executing function definition statements. Each example contains a definition statement and a description of how the statement affects the frame. The following examples are executed in the context of the following frame.

Each example is executed in the context of this frame, unaffected by previous examples:



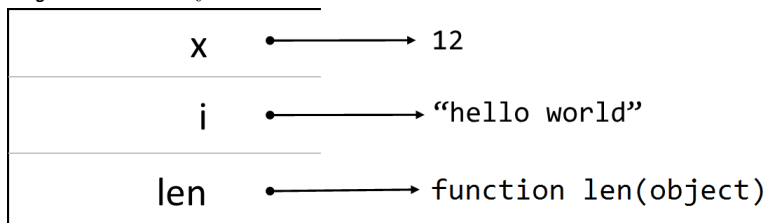
1. `def absVal(x):`
 BODY_STATEMENTS

The definition creates a new local variable `absVal` and assigns it to a function value with formal parameter `x` and body *BODY_STATEMENTS*.



2. `def len(object):`
 BODY_STATEMENTS

The definition reassigns the local variable `len` and assigns it to a function value with formal parameter `object` and body *BODY_STATEMENTS*.



8.2 Variable Access Expressions, Refined

This section replaces Section 4.1 with a more powerful set of rules.

Recall that in general, a variable access expression such as `x` has the form:

VAR_EXPR

Also, recall that Python uses a structure called a frame to keep track of the variables that currently exist. In fact, Python can access and assign variables in multiple frames. These frames are arranged in a structure called an *environment*, which is a list of frames.

Each frame consists of

- A set of bindings, each of which consists of a symbol (variable) bound to a value, and
- A reference to its “parent” frame.

In this course, the environment always has exactly two or three frames: either

- **global** → built-in (built-in is global’s parent), or
- **local** → global → built-in (global is local’s parent, built-in is global’s parent).

In either case, there is a “current” frame, where most of the work with variables is done. This is always the first element of the list — either the local or the global. The current frame is bolded in the list above.

The local frame is a result of function calls

When Python executes a function call, it sets the current frame to a new, empty local frame. The local frame’s parent is determined by where the function’s definition is. Every function definition in this course is at the module level, so you can assume that every local frame’s parent is the global. (You can define a function inside a function, which would make the local frame’s parent another local frame. This is powerful, but potentially confusing and is beyond the scope of this course.)

Note that a function’s environment is distinct from where the function is called. This means that calling function `b` from function `a` does not give `b` access to `a`’s variables. In general, this is a good thing — it means that functions you call can’t inadvertently modify your variables.

This new view of the Python environment complicates variable accesses and assignments. From now on, when interacting with variables, use the rules in the following sections for accesses and assignments.

8.2.1 Rules for Evaluation

This section replaces Section 4.1.1 with a more powerful set of rules.

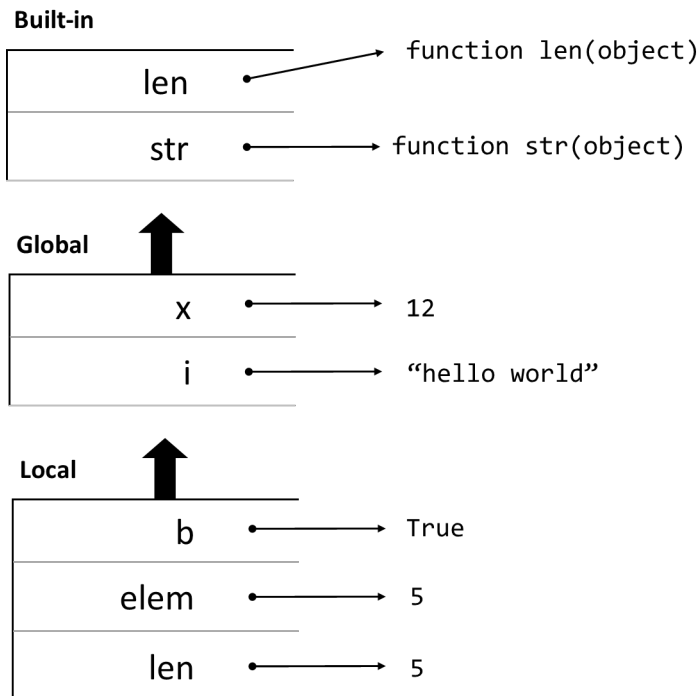
To evaluate a variable access expression,

1. Search the current frame for a binding for the variable *VAR_EXPR*. If you find the variable, replace the variable access with that variable’s value. Otherwise try the current frame’s parent. Continue until you either find the variable or reach the end of the environment.
2. If you reached the end of the environment without finding the variable, raise an error — the variable is not defined.

8.2.2 Examples

Below are examples of evaluating variable access expressions. Each example’s first line is the variable access expression, and the second is the value to which that expression evaluates.

Each example is executed in the context of this frame, unaffected by previous examples:



1. elem
5
2. i
"hello world"
After not finding `i` defined in the current frame, Python searched the subsequent frames in the environment until finding a binding for `i`.
3. o
ERROR. The variable `o` is not defined.
Python searched each frame in the environment, but none of them had a binding for `o`.
4. len
5

8.3 Variable Assignment Statements

This section describes the rarely-used `global` keyword, which enables you to reassign a variable in a different frame than the current one.

Recall that in general, a variable assignment statement such as `x = 18` has the form:

VAR_EXPR = EXPR

The `global` keyword

Even in an environment with multiple frames, all variable assignments happen in the current frame. Even if a variable with this name exists in a parent frame, a variable assignment always creates a new variable in the current frame and doesn't reassign the older variable.

This causes a problem when you want to reassign a variable in the global frame from inside a function. Instead of reassigning the global variable, the assignment creates a new variable of the same name in the local frame.

To assign this older variable instead of creating a new variable, precede any assignments (or accesses) to the variable with the `global` statement. The global statement follows the form:

```
global VAR_EXPR
```

Where *VAR_EXPR* is a variable in the global frame. After this statement, assigning *VAR_EXPR* reassigns it in the global frame, not the local one.

The `global` keyword does not affect variable access rules — only rules for reassignment.

8.3.1 Rules for Evaluation

To execute a variable assignment statement,

1. Evaluate *EXPR* to a value and replace the expression with that value.
2. Do one of the following:

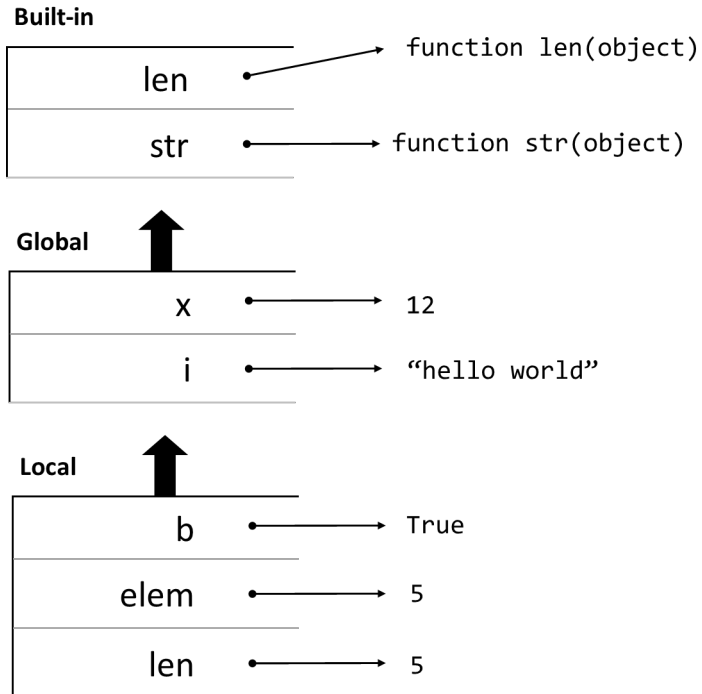
If the assignment to *VAR_EXPR* is preceded by `global VAR_EXPR` Create a new variable in the global frame called *VAR_EXPR* and give it the value from the previous step. If a variable with that name already exists in the global frame, overwrite its value with this new one.

Otherwise Create a new variable in the local frame called *VAR_EXPR* and give it the value from the previous step. If a variable with that name already exists in the local frame, overwrite its value with this new one.

8.3.2 Examples

Below are examples of executing variable assignment statements. Each example contains one or two Python statements and a description of how those statements affect the environment.

Each example is executed in the context of this frame, unaffected by previous examples:



1. `var = 18`
Creates `var` in local frame, sets value to 18.
2. `elem = x`
Updates `elem` in local frame, sets value to 12.
3. `x = 18`
Creates `x` in local frame, sets value to 18.
4. `len = str`
Updates `len` in the local frame, sets value to the value that `str` has.
5. `elem = elem + 1`
Updates `elem` in the local frame, sets value to 6.
6. `b = not b`
Updates `b` in the local frame, sets value to False.
7. `x = 17 + len`
Creates `x` in the local frame, sets value to 22.
8. `elem = elem + str`
ERROR. The plus operator can't operate on a function and an integer.
9. `b = b or False`
Updates `b` in the local frame, sets value to True.
10. `global i`
`i = 'a string'`
Updates `i` in the global frame, sets value to "a string".

8.4 Function Call Expressions

A function call expression invokes a function. Here are some examples of function calls:

```
str(17)      ⇒  "17"  
len([1, 2, 3]) ⇒  3  
abs(-1)     ⇒  1
```

In general, a function call has the form:

FUNC_EXPR (*PARAM_EXPR*, *PARAM_EXPR*, ..., *PARAM_EXPR*)

8.4.1 Rules for Evaluation

When you call a function, its body is evaluated in a new, empty frame. When the function returns, the new frame is discarded and the frame that was the current frame for the call site is once again made the current frame.

To evaluate a function call,

1. At the call site (where you call the function):
 - (a) Evaluate the function expression and replace it by the value.
 - (b) From left to right, evaluate each argument expression to a value and replace the expression with that value.
 - (c) If the value of the function expression is not a function value, raise an error. If the number of arguments (or the names, for keyword arguments) is not compatible with the function's declaration, raise an error. This usually requires that the number of actual arguments is the same as the number of formal parameters.
2. Create a new local frame for the called function. Make its parent the frame where the function is *defined* (not where it is called). Make the new frame be the current frame.
3. In the new local frame:
 - (a) Assign the actual argument values (the ones you evaluated at the call site) to the formal parameter variables (in the called function). Note that a formal parameter variable is always a new variable in the new frame, not a reuse of any existing variable of the same name.
 - (b) Evaluate the body of the called function. If you execute a **return** statement (of the form **return** *EXPR*), evaluate the expression to a value and remember the value (this is called the "return value"). If the **return** statement does not include an expression, then the return value is **None**. If you finish executing the body without executing a **return** statement, then the return value is **None**.
4. Discard the current frame, and make the previous current frame (at the call site) current again.
5. At the call site (where you call the function): The function call expression evaluates to the function's return value. Replace the function call expression by that value.

8.4.2 Examples

Below are examples of evaluating function call expressions. Each example contains a function call, the value of the function's formal parameters before executing the call, and the function's return value (the value to which the function call evaluates). In the following examples, the following functions are defined:

```
def absVal(x):
    if x < 0:
        return -x
    return x

# This definition overrides the built-in str function.
def str(x):
    return len(x)

# This definition of max is buggy, as shown in example 6 below.
def max(a, b):
    if a > b:
        return a
    elif a < b:
        return b

def list_contains_value(lst, val):
    for element in lst:
        if element == val:
            return True
    return False

def total_sum(lst, index):
    if index == len(lst):
        return 0
    else:
        return lst[index] + total_sum(lst, index + 1)
```

1. `absVal(-7)`
parameter values: x is -7
return value: 7
2. `absVal(8)`
parameter values: x is 8
return value: 8
3. `absVal(0)`
parameter values: x is 0
return value: 0
4. `str([1, 2, 3])`
parameter values: object is

list		
1	2	3

return value: 3
5. `max(17, -8)`
parameter values: a is 17, b is -8
return value: 17

6. `max(17, 17)`
parameter values: a is 17, b is 17
return value: `None`
7. `list_contains_value([1, 2, 3], 4)`
parameter values: `lst` is

1	2	3
---	---	---

, `val` is 4
return value: `False`
8. `list_contains_value([], 0)`
parameter values: `lst` is

--

, `val` is 0
return value: `False`
9. `list_contains_value(['a', 'b'], 'a')`
parameter values: `lst` is

"a"	"b"
-----	-----

, `val` is "a"
return value: `True`
10. `total_sum([], 0)`
parameter values: `lst` is

--

, `index` is 0
return value: 0
11. `total_sum([1], 0)`
parameter values: `lst` is

1

, `index` is 0
return value: 1
12. `total_sum([1, 2, 3], 0)`
parameter values: `lst` is

1	2	3
---	---	---

, `index` is 0
return value: 6
13. `total_sum([], 1)`
parameter values: `lst` is

--

, `index` is 1
ERROR: list index out of range