UDACITY Nanodegree Catalog Joanne Dorning

Intro to Computer Science

cs101 »

View History

Contents

Arithmetic Expressions

addition: <***Number***> + <***Number***> = <***Number***>

- outputs the sum of the two input numbers

multiplication: <***Number***> * <***Number***> = <***Number***>

- outputs the product of the two input numbers

subtraction: <***Number***> - <***Number***> = <***Number***>

- outputs the difference between the two input numbers

division: <***Number***> / <***Number***> = <***Number***>

- outputs the result of dividing the first number by the second **Note**: if both numbers are whole numbers, the result is truncated to just the whole number part.

Variables and Assignment

Names

A ***<name>*** in Python can be any sequence of letters, numbers, and underscores (_) that does not start with a number. We usually use all lowercase letters for variable names, but capitalization must match exactly. Here are some valid examples of names in Python (but most of these would not be good choices to actually use in your programs):

- my_name
- one2one
- Dorina
- this_is_a_very_long_variable_name

Assignment Statement

An assignment statement assigns a value to a variable:

- *<name> = <expression>*

After the assignment statement, the variable ***<name>*** refers to the value of the ***<expression>*** on the right side of the assignment. An ***<expression>*** is any Python construct that has a value.

Multiple Assignment

We can put more than one name on the left side of an assignment statement, and a corresponding number of expressions on the right side:

- ***<name1>, <name2>, ... = <expression1>, <expression2>, ...***

All of the expressions on the right side are evaluated first. Then, each name on the left side is assigned to reference the value of the corresponding expression on the right side. This is handy for swapping variable values. For example,

- **s, t = t, s**

would swap the values of s and t so after the assignment statement s now refers to the previous value of **t**, and **t** refers to the previous value of **s**.

Note: what is really going on here is a bit different. The multiple values are packed in a tuple (which is similar to the list data type introduced in Unit 3, but an immutable version of a list), and then unpacked into its components when there are multiple names on the left side. This distinction is not important for what we do in CS101, but does become important in some contexts.

Procedures

A **procedure** takes inputs and produces outputs. It is an abstraction that provides a way to use the same code to operate on different data by passing in that data as its inputs.

Defining a procedure:

```
def <Name>(<Parameters>):
      <Block>
```

The **<*Parameters*>** are the inputs to the procedure. There is one **<*Name*>** for each input in order, separated by commas. There can be any number of parameters (including none).

To produce outputs:

```
return <Expression>, <Expression>, ...
```

There can be any number of expressions following the return (including none, in which case the output of the procedure is the special value **None**).

Using a procedure:

```
<''Procedure''>(<Input>, <Input>, ...)
```

The number of inputs must match the number of parameters. The value of each input is assigned to the value of each parameter name in order, and then the block is evaluated.

If Statements

The **if** statement provides a way to control what code executes based on the result of a test expression.

```
if <TestExpression>:
   <Block>
```

The code in **<*Block*>** only executes if the **<*TestExpression*>** has a **True** value.

Alternate clauses. We can use an **else** clause in an **if** statement to provide code that will run when the <TestExpression> has a False value.

```
if <TestExpression>:
   <BlockTrue>
else:
        <BlockFalse>
```

Logical Operators

The **and** and **or** operators behave similarly to logical conjunction (and) and disjunction (or). The important property they have which is different from other operators is that the second operand expression is evaluated only when necessary.

- ***<Expression~1~> and <Expression~2~>***
- If **Expression~1~** has a **False** value, the result is **False** and ***Expression~2~*** is not evaluated (so even if it would produce an error it does not matter). If ***Expression~1~*** has a **True** value, the result of the **and** is the value of ***Expression~2~***.
- ***<Expression~1~>*** or ***<Expression~2~>***
- If ***Expression~1~*** has a **True** value, the result is **True** and ***Expression~2~*** is not evaluated (so even if it would produce an error it does not matter). If ***Expression~1~*** has a **False** value, the result of the **or** is the value of ***Expression~2~***'.

Loops

Loops provide a way to evaluate the same block of code an arbitrary number of times.

While Loops

A **while** loop provides a way to keep executing a block of code as long as a test expression is **True**.

```
        while <TestExpression>:
            <Block>
```

If the <*TestExpression*> evaluates to **False**, the **while** loop is done and execution continues with the following statement. If the <*TestExpression*> evaluates to **True**, the <*Block*> is executed. Then, the loop repeats, returning to the <*TestExpression*> and continuing to evaluate the <*Block*> as long as the <*TestExpression*> is **True**.

For Loops

A **for** loop provides a way to execute a block once for each element of a collection:

```
for <Name> in <Collection>:
    <Block>
```

The loop goes through each element of the collection, assigning that element to the **<**Name> and evaluating the **<**Block>. The collection could be a String, in which case the elements are the characters of a string; a List, in which case the elements are the elements of the list; a Dictionary, in which case the elements are the keys in the dictionary; or many other types in Python that represent collections of objects.

Strings

A **string** is sequence of characters surrounded by quotes. The quotes can be either single or double quotes, but the quotes at both ends of the string must be the same type. Here are some examples of strings in Python:

- "silly"
- 'string'
- "I'm a valid string, even with a single quote in the middle!"

String Operations

length: **len(<**String>*)* = <Number>

- Outputs the number of characters in **<**String>

string concatenation: **<**String> + <String> = <String>

- Outputs the concatenation of the two input strings (pasting the string together with no space between them)

string multiplication: **<**String> * <Number> = <String>

- Outputs a string that is <number*> copies of the input <string> pasted together*

INDEXING STRINGS

The indexing operator provides a way to extract subsequences of characters from a string.

string indexing: <*String*>[<*Number*>] = <*String*>

- Outputs a single-character string containing the character at position <number> of the input <string>. Positions in the string are counted starting from **0**, so **s[1]** would output the second character in **s**. If the <*Number*> is negative, positions are counted from the end of the string: **s[-1]** is the last character in **s**.*

string extraction: <*String*>[<*Start Number*>:<*Stop Number*>] = <*String*>

- Outputs a string that is the subsequence of the input string starting from position <*Start Number*> and ending just before position <*Stop Number*>. If <*Start Number*> is missing, starts from the beginning of the input string; if <*Stop Number*> is missing, goes to the end of the input string.

find The **find** method provides a way to find sub-sequences of characters in strings.

find: <*Search String*>.find(<*Target String*>) = <*Number*>

- Outputs a number giving the position in <*Search String*> where <*Target String*> first appears. If there is no occurrence of <*Target String*> in <*Search String*>, outputs **-1**.

To find later occurrences, we can also pass in a number to find:

find after: <*Search String*>.find(<*Target String*>, <*Start Number*>) = <*Number*>

- Outputs a number giving the position in <*Search String*> where <*Target String*> first appears that is at or after the position give by <*Start Number*>. If there is no occurrence of <*Target String*> in <*Search String*> at or after <*Start Number*>, outputs **-1**.

CONVERTING BETWEEN NUMBERS AND STRINGS

str: str(<*Number*>) = <*String*>

- Outputs a string that represents the input number. For example, **str(23)** outputs the string '**23**'.

ord: ord(<*One-Character String*>) = <*Number*>

- Outputs the number corresponding to the input string.

chr: chr(<*Number*>) = <*One-Character String*>

- Outputs the one-character string corresponding to the number input. This function is the inverse of **ord: chr(ord(a)) = a** for any one-character string a.

SPLITTING STRINGS

split: <*String*>.split() = [<*String*>, <*String*>, ...]

- outputs a list of strings that are (roughly) the words contained in the input string. The words are determined by whitespace (either spaces, tabs, or newlines) in the string. (We did not cover this in class, but split can also be used with an optional input that is a list of the separator characters, and a second optional input that controls the maximum number of elements in the output list.)

LOOPING THROUGH STRINGS

A **for** loop provides a way to execute a block once for each character in a string (just like looping through the elements of a list):

```
for <*Name*> in <*String*>:
            <*Block*>
```

The loop goes through each character of the string in turn, assigning that element to the <*Name*> and evaluating the <*Block*>.

Lists

A list is a mutable collection of objects. The elements in a list can be of any type, including other lists.

Constructing a list. A list is a sequence of zero or more elements, surrounded by square brackets:

- **[<*Element*>, <*Element*>, ...]**

Selecting elements: <*List*>[<*Number*>] = <*Element*>

- Outputs the value of the element in <*List*> at position <*Number*>. Elements are indexed starting from 0.

Selecting sub-sequences: <*List*>[<*Start*> : <*Stop*>] = <*List*>

- Outputs a sub-sequence of <*List*> starting from position <*Start*>, up to (but not including) position <*Stop*>.

Update: <*List*>[<*Number*>] = <*Value*>

- Modifies the value of the element in <*List*> at position <*Number*> to be <*Value*>.

Length: len(<*List*>) = <*Number*>

- Outputs the number of (top-level) elements in <*List*>.

Append: <*List*>.append(<*Element*>)

- Mutates <*List*> by adding <*Element*> to the end of the list.

Concatenation: *<List~1~> + <List~2~> = <List~3~>***

- Outputs a new list that is the elements of <*List~1~*> followed by the elements of <*List~2~*>.

Popping: <*List*>.pop() = <*Element*>

- Mutates <*List*> by removing its last element. Outputs the value of that element. If there are no elements in <*List*>, [].pop() produces an error.

Finding: <*List*>.index(<*Value*>) = <*Number*>

- Outputs the position of the first occurrence of an element matching <*Value*> in <*List*>. If <*Value*> is not found in <*List*>, produces an error.

Membership: <*Value*> in <*List*> = <Boolean>

- Outputs **True** if <*Value*> occurs in <*List*>. Otherwise, outputs **False**.

Non-membership: <*Value*> not in <*List*> = <*Boolean*>

- Outputs **False** if <*Value*> occurs in <*List*>. Otherwise, outputs **True**.

Loops on Lists

A **for** loop provides a way to execute a block once for each element of a list:

```
for <Name> in <List>:
    <Block>
```

This page was last edited on 2016/08/14 20:51:16.

POPULAR NANODEGREE PROGRAMS +

STUDENT RESOURCES +

UDACITY +

INQUIRIES +