Python Reference

cs101: Unit 2 (includes Unit 1)

Arithmetic Expressions

addition: ***<Number> + <Number>***
outputs the sum of the two input numbers
multiplication: ***<Number> * <Number>***
outputs the product of the two input numbers
subtraction: ***<Number> - <Number>***
outputs the difference between the two input numbers
division: ***<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.
modulo: ***<Number> % <Number>***
outputs the remainder of dividing the first number by the second

Comparisons

equality: ***<Value> == <Value>***
outputs **True** if the two input values are equal, **False** otherwise
inequality: ***<Value> != <Value>***
outputs **True** if the two input values are not equal, **False** otherwise
greater than: ***<Number$_1$> > <Number$_2$>***
outputs **True** if ***Number$_1$*** is greater than ***Number$_2$***
less than: ***<Number$_1$> < <Number$_2$>***
outputs **True** if ***Number$_1$*** is less than than ***Number$_2$***
greater or equal to: ***<Number$_1$> >= <Number$_2$>***
outputs **True** if ***Number$_1$*** is not less than ***Number$_2$***
less than or equal to: ***<Number$_1$> <= <Number$_2$>***
outputs **True** if ***Number$_1$*** is not greater than ***Number$_2$***

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:

$<Name_1>, <Name_2>, \ldots = <Expression_1>, <Expression_2>, \ldots$

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.

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 Concatenation

We can use the **+** operator on strings, but it has a different meaning than when it is used on numbers.

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

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

We can also use the multiplication operator on strings:

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

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*>]

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>*]**

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*>)**

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*>)**

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**.

str

(Introduced in the last homework question, not in the lecture.)

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

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

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*>:
    <$Block_{True}$>
else:
    <$Block_{False}$>

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 $Expression_2$.

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**.

Break Statement

A **break** statement in the **<*Block*>** of a **while** loop, jumps out of the containing while loop, continuing execution at the following statement.

break