notebook_3_expressions_and_functions

March 21, 2018

1 Expressions and Functions

1.1 Objectives

1. Introduce basic Python expressions for transforming data
    1. Arithmetic
    2. Boolean
2. Use the conditional expression for situational expressions
3. Introduce the `lambda` expression for reusing expressions
4. Writing more complicated functions with the `def` statement

1.2 Expressions and Statements

- Two types of Python code
- **statements** "Do this"
    - Does not return/evaluate to a value
    - Often contain expressions
- **expressions** "Compute this value"
    - Allows returns data
- I prefer expressions

```
In [1]: sum([1,2,3]) # expression (returns 6)
```

```
Out[1]: 6
```

```
In [3]: x = sum([1,2,3]) # assignment statement (no value)
```

```
In [4]: x # Expression (returns value assigned to x)
```

```
Out[4]: 6
```

1.3 Arithmetic operations

```
In [7]: 2 + 3.5
```

```
Out[7]: 5.5
```

```
In [8]: 2 - 3.5
```

```
Out[8]: -1.5
```

```
In [9]: 2*3.5
```

```
Out[9]: 7.0
```

```
In [10]: 2/3 # True division
```

```
Out[10]: 0.6666666666666666
```

```
In [12]: 10//3 # Integeer division
```

```
Out[12]: 3
```

```
In [14]: 10 % 3 # Mod/remainder
```

```
Out[14]: 1
```

```
In [15]: 2 ** 3 # Powers!
```

```
Out[15]: 8
```

```
In [16]: 2 ^ 3 # BEWARE: bitwise exclusive or
```

```
Out[16]: 1
```

1.3.1 Exercise 1

Investigate which operations can be applied to two strings, say `"a"` and `"b"`.

```
In [ ]:
```

1.3.2 Exercise 2

Investigate various operations on 2 and `"a"`

```
In [ ]:
```

1.4 Boolean operations

- **Boolean expressions** evaluate to True or False
- **Boolean operations** answer Yes/No questions

1.5 Comparison operators

```
In [40]: (2 == 3, 2 < 3, 2 <= 3, 2 != 3)
```

```
Out[40]: (False, True, True, True)
```

1.6 Container operations

```
In [41]: 2 in [1,2,3]
```

```
Out[41]: True
```

```
In [42]: "a" in "Todd"
```

```
Out[42]: False
```

```
In [43]: "t" in "Todd"
```

```
Out[43]: False
```

```
In [44]: 2 in [1, [2, 3]]
```

```
Out[44]: False
```

1.7 Combining boolean expressions

- Use `and`, `or`, and `not`

```
In [51]: x = 5
         name = "Todd"
         x % 2 == 1 and "a" in "Todd"
```

```
Out[51]: False
```

```
In [52]: x % 2 == 1 or "a" in "Todd"
```

```
Out[52]: True
```

```
In [49]: not 2 < 3
```

```
Out[49]: False
```

1.8 The `lambda` expression

- Allows reuse of expressions
- Variables become **Parameters**
  - insert values later

1.9 The `lambda` expression

- The *value* of a `lambda` expression is a function
  - Use function calls to execute
- Syntax: 'lambda parameter(s): expression
  - Can only hold one expression
  - No statements
  - Always returns the value

```
In [62]: f = lambda x: x**2
         type(f)
```

```
Out[62]: function
```

```
In [64]: f(3)
```

```
Out[64]: 9
```

```
In [65]: g = lambda x, y: x**y
         g(2,4)
```

```
Out[65]: 16
```

1.10 Reusing expressions

- Replace values with variables
- Identify variables that might change
- Make a function with the **lambda expression**
  - changeable variables are **expressions**

```
In [66]: # Original Computatation
         12*15*0.5625
```

```
Out[66]: 101.25
```

```
In [67]: # Substitute names for values to add meaning
         length = 12
         width = 15
         tile_conv = 0.5625
         num_tile = length*width*tile_conv
         num_tile
```

```
Out[67]: 101.25
```

Identify the variables that might change length* width*tile_conv

```
In [15]: tile_conv = 0.5625
         num_tile = lambda length, width: length*width*tile_conv
         num_tile(12,15)
```

```
Out[15]: 101.25
```

1.11 How to read a lambda expression

- `lambda` means execute later
  - does nothing now
- Parameter(s) are "hole(s)" in the expression
  - value filled in later

1.11.1 The `def` statement

- Starts with `def`
- First line is header
  - name
  - parameters (in parentheses)
  - ends in :
    - to open a code block
- Body
  - Indented 4 spaces
  - Requires explicit `return`

```
In [1]: def sum_sqr(x, y):
            """ Square and add two numbers"""
            output = x**2 + y**2
            return output
```

1.12 Docstrings

- Docstring: String on the first line of a `def` statement
  - Usually multiline
- doc strings are used to document functions
  - What is shown when calling help

```
In [1]: def sum_sqr(x, y):
            """ add the squares of two numbers"""
            output = x**2 + y**2
            return output

In [2]: help(sum_sqr)

Help on function sum_sqr in module __main__:

sum_sqr(x, y)
    add the squares of two numbers
```

1.13 Writing more complicated function with `def`

- Allows multiple lines
- Allows statements
- Must explicitly `return` a value

```
In [3]: def leap_year(year):
            """ Determine if a given year (int) is a leap year"""
            if year % 400 == 0:
                return True
            elif year % 100 == 0:
                return False
            elif year % 4 == 0:
                return True
            else:
                return False

In [4]: leap_year(2000)

Out[4]: True

In [5]: leap_year(1900)

Out[5]: False
```

1.14 The `assert` statement

- Syntax: `assert expression`
- Silent if expression is true
- Error if expression is false
- Include and optional message

```
In [6]: assert leap_year(2000) == True
        assert leap_year(1900) == False

In [7]: assert leap_year(1900) == True, "Centuries not divisible by 4 are not leap years"


        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-7-d640e9015aaa> in <module>()
    ----> 1 assert leap_year(1900) == True, "Centuries not divisible by 4 are not leap years"


        AssertionError: Centuries not divisible by 4 are not leap years
```

1.15 Writing test functions

- Name the function "test_func_name"
  - Allows automatic testing with py.test module
- Body consists of assert statements
- Follow with a function call
- No errors == passed the test

```
In [12]: def test_leap_year():
             assert leap_year(2000) == True
             assert leap_year(1902) == False
             assert leap_year(2004) == True
             assert leap_year(1900) == False
         test_leap_year() # Silence is golden!
```

1.15.1 Converting `lambda` expression to `def` statements

- lambda expressions process one expression
  - add one assignment statement
- lambda expressions always return the value
  - add a return statement
- Always test on some examples!

```
In [13]: tile_conv = 0.5625
         num_tile = lambda length, width: length*width*tile_conv
         num_tile(10,12)

Out[13]: 67.5

In [14]: def num_tile(length, width):
             """Compute the number of 9 inch
                tile for a room of area length*width"""
             output = length*width*tile_conv
             return output

         def test_num_tile():
             assert num_tile(10, 12) == 67.5
             assert num_tile(0, 12) == 0
         test_num_tile()
```

1.15.2 Exercise 3

Many people keep time using a 24 hour clock (11 is 11am and 23 is 11pm, 0 is midnight). If it is currently 13 and you set your alarm to go off in 50 hours, it will be 15 (3pm). Write a Python function to solve the general version of the above problem using variables and functions.

Hint You will want to use modular arithmetic

Step 1 - Solve a specific example with variables and expressions

In [ ]:

Step 2 - Convert your expression to a general solution with a `lambda` expression

In [ ]:

Step 3 - Convert your `lambda` expression to a function (remember to return)

In [ ]:

Step 4 - Clean up your code by adding a docstring and test function

In [ ]:

1.15.3 Exercise 4

Write a function that will compute the area of a circle. Use variable and lambda expressions in your solution.

Hint Be sure to use `math.pi` for precision.

Step 1 - Solve a specific example with variables and expressions

In [ ]:

Step 2 - Convert your expression to a general solution with a function

In [ ]:

Step 3 - Convert your `lambda` expression to a function (remember to return)

In [ ]:

Step 4 - Clean up your code by adding a docstring and test function

In [ ]: