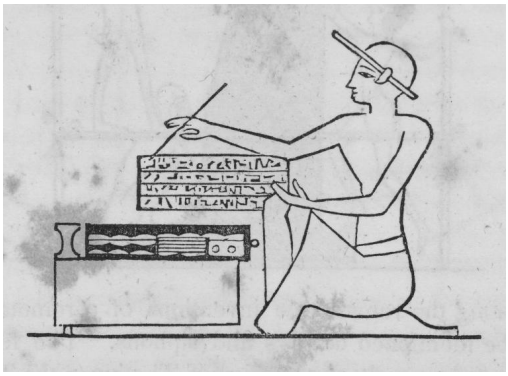

# CPSC 5061—Programming I for Business

## Worksheet 9

### Creating Functions

*Technique*

**Start with a working program, then make it more useful by abstracting it and turning it into a function.**

1. Come up with a good name for the function.
2. Determine which variables should be parameters, if any.
3. Determine what the return value(s) should be, if any.
4. Do the syntactic transformation to achieve #1 - #3, by adding a function header line, indenting the body, adding a return, and putting in a suitable docstring.
5. Test it with the original values of the parameters and also with new values.

*Example: Pure Function*

Given this program that computes the net amount of a sale:

```
purchase = 489.50  # in dollars
tax_rate = 10.1  # percent
tax = round(purchase * tax_rate / 100.0, 2)
net = purchase + tax
print('Net amount:', net)
```

Let's follow the steps in the above technique:

1. A good name is **net_sale**.
2. Think about which items might change under different circumstances and what would be most useful to clients as parameters. In this case, we could choose just **purchase**, but by adding **tax_rate** also as a parameter, the function is even more general.
3. The net amount of the sale, **net**, is the purpose of this program so that is what we should return. We want to return the result and not print it. The client can print it if they want to.
4. Do it:

```
def net_sale(purchase, tax_rate):
    """Calculate net purchase price after adding tax.
    Expects purchase in dollars and tax_rate as percent.
    """
    tax = round(purchase * tax_rate / 100.0, 2)
    net = purchase + tax
    return net
```

5. Test it:

```
>>> net_sale(489.50, 10.1)  # original values of parameters
538.94
>>> net_sale(1000.00, 10.0)  # other values
1100.0
>>> net_sale(0.00, 10.0)  # how about zero?
0.0
>>> net_sale(-489.50, 10.1)  # what about a refund?
-538.94
>>> help(net_sale)  # check the docstring too!
Help on function net_sale in module __main__:

net_sale(purchase, tax_rate)
    Calculate net purchase price after adding tax.
    Expects purchase in dollars and tax_rate as percent.
```

*Using Default Arguments*

When moving a program to a function, some of the would-be parameters might be something that we almost always want to be the same value. In these cases, make sure to put these parameters on the end of the parameter list and add default values. Then then client callers can exclude them and get the usual values for them.

Continuing on with the above example, we might think that in Seattle, the tax is almost always 10.1%, so that should be the default and we would only have to override it if it were different.

```
def net_sale(purchase, tax_rate=10.1):
    """Calculate net purchase price after adding tax.
    Expects purchase in dollars and tax_rate as percent.
    """
    tax = round(purchase * tax_rate / 100.0, 2)
    net = purchase + tax
    return net
```

Then we run the same tests as above, but also add some more where we exclude the defaulted parameters.

```
>>> net_sale(1000.00, 10.0)  # as before
1100.0
>>> net_sale(0.00, 10.0)  # as before
0.0
>>> net_sale(489.50)  # use default value of 10.1%
538.94
>>> net_sale(1000.00)
1101.0
```

### *Function Composition*

Many times, we want the individual pieces of our program to be separate functions and the functions to call each other as necessary.

In the above example, we might notice that the net sales number is very useful, but when printing the invoice, we'd like to also state the sales tax separately, too. So, we might want a function that just calculates sales tax. That function can then be used by the net_sale function, too. We call this untangling of the functionality into separate pieces *refactoring*. In the above example, we might end up with the following pair of functions:

```
def sales_tax(purchase, tax_rate=10.1):
    """Calculate the tax due for a given sale.
    Expects purchase in dollars and tax_rate as percent.
    """
    return round(purchase * tax_rate / 100.0, 2)

def net_sale(purchase, tax_rate=10.1):
    """Calculate net purchase price after adding tax.
    Expects purchase in dollars and tax_rate as percent.
    """
    return purchase + sales_tax(purchase, tax_rate)
```

Note that both of them have the **tax_rate** parameter with a default value. This is because we anticipate client callers that would call both and would like to use the default rate for each. But when **net_sale** calls the **sales_tax** it has to pass the **tax_rate** that it is using, no matter what it is.

Here's an example client. It is a function that prints an invoice for a customer:

```
def print_bellevue_invoice(customer, item, price):
    """Prints an invoice for given customer purchasing
    the one given item within Bellevue, WA (tax rate: 10.0%).
    """
    tax_rate = 10.0
    city = 'Bellevue, WA'
    print('INVOICE')
    print('Store Location:', city)
    print('Customer:', customer, '\n\n')
    print('Qty  Item                              Price')
    print('1    %-25s $%12.2f' % (item, price))
    print()
    print('%30s  %12.2f' % ('Tax', sales_tax(price, tax_rate)))
    print('%30s $%12.2f' % ('Total', net_sale(price, tax_rate)))
```

And a test for the above:

```
>>> print_bellevue_invoice('Kevin', 'Blue Suit', 500.00)
INVOICE
Store Location: Bellevue, WA
Customer: Kevin


Qty  Item                          Price
1    Blue Suit                $      500.00

                            Tax          50.00
                          Total $      550.00
```

*Exercise*

Turn the following program into a function. Show all five steps.

```
s = 'Mean body temp for 4 subspecies ranges from 25.2 - 27.8C.'
count = 0
for c in s:
    if c.isdigit():
        count += 1
print(count)
```

1. Function Name:

2. Parameters:

3. Return Value:

4. Write the code (including docstring):

5. Test it: