Introduction to Programming with Python

Qitian Liao

Aug 3, 2020

Contents

1 Functions

1.1 Elements of Programming

Every programming language has three basic mechanisms:
Primitive expressions and statements: This represents the simplest building blocks that the language provides
Means of combination: Compound elements are built from simpler ones
Means of abstraction: Compound elements can be named and manipulated as units.

1.2 Expressions

An expression describes a computation and evaluates to a value.
Primitive Expressions:

1. Number or Numeral. More precisely, the expression that you type consists of the numerals that represent the number in base 10. Expressions representing numbers may be combined with mathematical operators to form a compound expression, which the interpreter will evaluate.

```
>>> 13 + 13
26
```

   These mathematical expressions use infix notation, where the operator (e.g., +, -, *, or /) appears in between the operands (numbers). Python includes many ways to form compound expressions.

2. Name

```
>>> max
<built-in function max>
```

3. String

```
>>> "about"
'about'
```

1.3 Call Expressions

The most important kind of compound expression is a call expression, which applies a function to some arguments.

```
>>> max(6, 8)
8
```

$$\text{Operator}(\text{Operand}, \text{Operand})$$

This call expression has subexpressions: the operator is an expression that precedes parentheses, which enclose a comma-delimited list of operand expressions. The operator specifies a function. When this call expression is evaluated, the function max is called with arguments 6 and 8, and returns a value of 8.

Evaluation procedure for call expressions

1. Evaluate the operator and then the operand subexpressions.

2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpression.

The order of the arguments in a call expression matters. For instance, the function pow raises its first argument to the power of its second argument.

```
>>> pow(2, 5)
32
>>> pow(5, 2)
25
```

Function notation has three principal advantages over the mathematical convention of infix notation.

1. Functions may take an arbitrary number of arguments.

   ```
   >>> max(1, 2, 3, 4)
   4
   ```

2. Function notation extends in a straightforward way to nested expressions, where the elements are themselves compound expressions. In nested call expressions, unlike compound infix expressions, the structure of the nesting is entirely explicit in the parentheses.

   ```
   >>> max(min(1, -2), min(pow(3, 5), -4))
   -2
   ```

3. Mathematical notation has a great variety of forms: multiplication appears between terms, exponents appear as superscripts, division as a horizontal bar, and a square root as a roof with slanted siding. All of this complexity can be unified via the notation of call expressions. While Python supports common mathematical operators using infix notation (like + and -), any operator can be expressed as a function with a name.

1.4 Importing Library Functions

Python defines a very large number of functions, including the operator functions mentioned in the preceding section, but does not make all of their names available by default. Instead, it organizes the functions and other quantities that it knows about into modules, which together comprise the Python Library. To use these elements, one imports them.
For example, the math module provides a variety of familiar mathematical functions:

```
from math import sqrt
>>> sqrt(25)
5
```

The operator module provides access to functions corresponding to infix operators:

```
from operator import add, sub, mul
>>> add(3, 5)
8
```

An import statement designates a module name (e.g., math), and then lists the named attributes of that module to import (e.g., sqrt). Once a function is imported, it can be called multiple times.

There is no difference between using these operator functions (e.g., add) and the operator symbols themselves (e.g., +). Conventionally, most programmers use symbols and infix notation to express simple arithmetic.

1.5 Exercises

Evaluate the following equations:

```
>>> max(min(3, -2), min(pow(2, 5), -4))
```

2 Names, Assignment and User-Defined Functions

2.1 Names

A critical aspect of a programming language is the means it provides for using names to refer to computational objects. If a value has been given a name, we say that the name binds to the value.

2.2 Assignment

In Python, we can establish new bindings using the assignment statement, which contains a name to the left of = and a value to the right:

```
>>> a = 10
>>> a
10
>>> a + 2
12
```

The = symbol is called the assignment operator in Python. Assignment is our simplest means of abstraction, for it allows us to use simple names to refer to the results of compound operations. In this way, complex programs are constructed by building, step by step, computational objects of increasing complexity.

Environment: The possibility of binding names to values and later retrieving those values by name means that the interpreter must maintain some sort of memory that keeps track of the names, values, and bindings. This memory is called an environment.

Names can also be bound to functions. For instance, the name max is bound to the max function we have been using. Functions, unlike numbers, are tricky to render as text, so Python prints an identifying description instead, when asked to describe a function.

```
>>> max
<built-in function max>
```

We can use assignment statements to give new names to existing functions.

```
>>> f = max
>>> f
<built-in function max>
>>> f(1, 2, 3)
3
```

And successive assignment statements can rebind a name to a new value.

```
>>> f = 2
>>> f
2
```

In Python, names are often called variable names or variables because they can be bound to different values in the course of executing a program. When a name is bound to a new value through assignment, it is no longer bound to any previous value. One can even bind built-in names to new values.

```
>>> max = 5
>>> max
```

```
5
```

After assigning max to 5, the name max is no longer bound to a function, and so attempting to call max(2, 3, 4) will cause an error.

```
>>> max(1, 2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

When executing an assignment statement, Python evaluates the expression to the right of = before changing the binding to the name on the left. Therefore, one can refer to a name in right-side expression, even if it is the name to be bound by the assignment statement.

```
>>> x = 2
>>> x = x + 1
>>> x
3
```

We can also assign multiple values to multiple names in a single statement, where names on the left of = and expressions on the right of = are separated by commas.

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
```

With multiple assignment, all expressions to the right of = are evaluated before any names to the left are bound to those values. As a result of this rule, swapping the values bound to two names can be performed in a single statement.

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

2.3 The Non-pure Print Function

The special **None** represents nothing in Python. A function that does not explicitly return a value will return **None**. **None** is not displayed by the interpreter as the value of an expression.

Pure functions: Functions have some input (their arguments) and return some output (the result of applying them). The built-in function abs can be depicted as a small machine that takes input and produces output. From the output below, we can see that the function **abs** is pure.

```
>>> abs(-1)
1
```

Properties of pure functions:

1. Applying them has no effects beyond returning a value.

2. Must always return the same value when called twice with the same arguments.

Non-pure functions: In addition to returning a value, applying a non-pure function can generate side effects, which make some change to the state of the interpreter or computer. A common side effect is to generate additional output beyond the return value, using the **print** function.

```
>>> print(1, 2, 3)
1 2 3
```

While **print** and **abs** may appear to be similar in these examples, they work in fundamentally different ways. The value that **print** returns is always **None**.The interactive Python interpreter does not automatically print the value **None**. In the case of print, the function itself is printing output as a side effect of being called.
A nested expression of calls to print highlights the non-pure character of the function.

```
>>> print(print(1), print(2))
1
2
None None
```

Let us see another example.

```
>>> two = print(2)
2
>>> print(two)
None
```

2.4 Exercises

For each of the expressions below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error", but include all output displayed before the error. If a function is displayed, write "Function".
Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is **None**.

```
>>> print(1, print(2))
```

```
>>> print(None, print(None))
```

```
>>> print(print(print(2)), print(3))
```

```
>>> print(4, 5) + 1
```

3 Control

3.1 Defining Functions

Function definition is a more powerful means of abstraction, in which it binds names to expressions.

```
def <name>(<formal parameters>):
   return <return expression>
```

Execution procedure for def statements:

1. Create a function with signature <name>(<formal parameters>). Function signature indicates how many arguments a function takes.
2. Set the body of that function to be everything indented after the first line. Function body defines the computation performed when the function is applied.
3. Bind <name> to that function in the current frame.

3.1.1 Environmental Diagrams

1. Every expression is evaluated in the context of an environment. Names have no meaning without an environment.
2. An environment is a sequence of frames, starting with the global frame and followed by a sequence of local frames.
3. A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Look at the following example.

```
>>> from operator import mul
>>> def square(x):
      return mul(x, x)
>>> square(square(3))
```

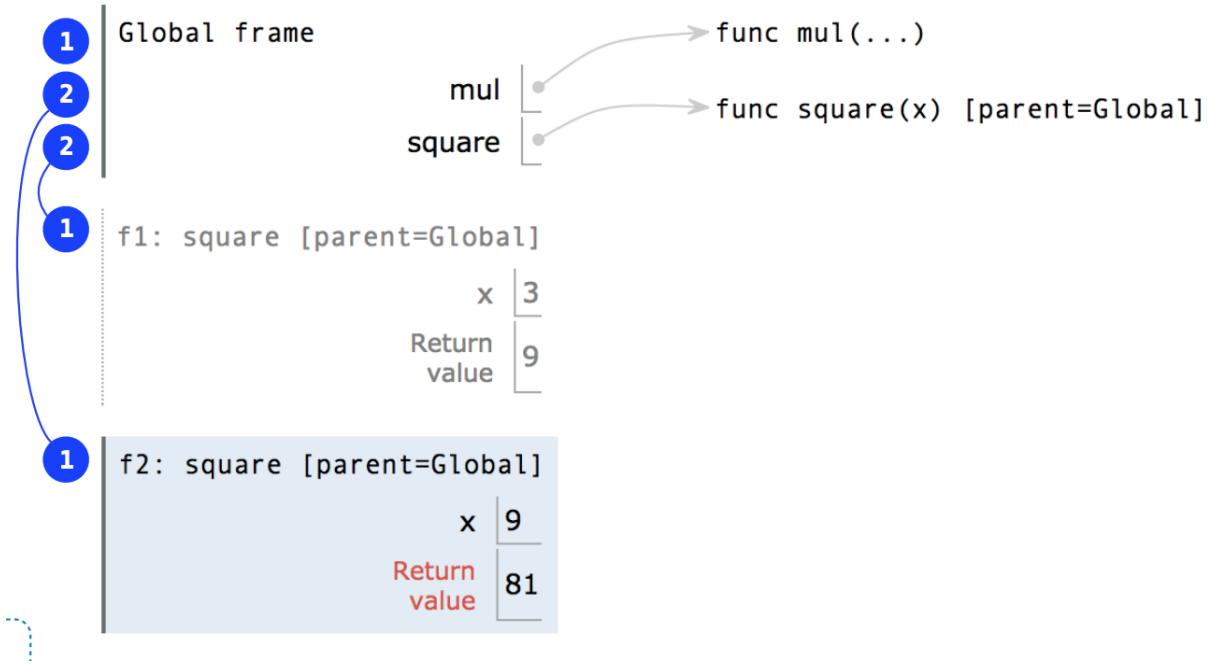
The corresponding environmental diagram is on the next page.

Sometimes we will have very complicated function calls, and environmental diagrams will come in very handy at that time. Look at the following example. (credit to CS61A Midterm Spring 2015). We will try to solve this after we cover higher order functions.

```
batman, superman, ivy = 1, -2, -3
def nanana(batman):
   while batman(superman) > ivy:
      def batman(joker):
         return ivy
   return -ivy

def joker(superman):
   if superman(batman):
      ivy = -batman
   return nanana

joker(abs)(abs)
```



3.1.2 Life Cycle of a User-Defined Function

1. Def Statement: A new function is created. Name bound to that function is in the current frame.
2. Call Expression: Operator and operands are evaluated. Function (value of operator) called on arguments (value of operands).
3. Calling/Applying: A new frame is created. Parameters are bound to the arguments. Body is executed in that new environment.

3.2 Conditional Statement

A statement is executed by the interpreter to perform an action.

3.2.1 Compound Statements

Above we have seen how we can define functions. And def statements are compound statements.

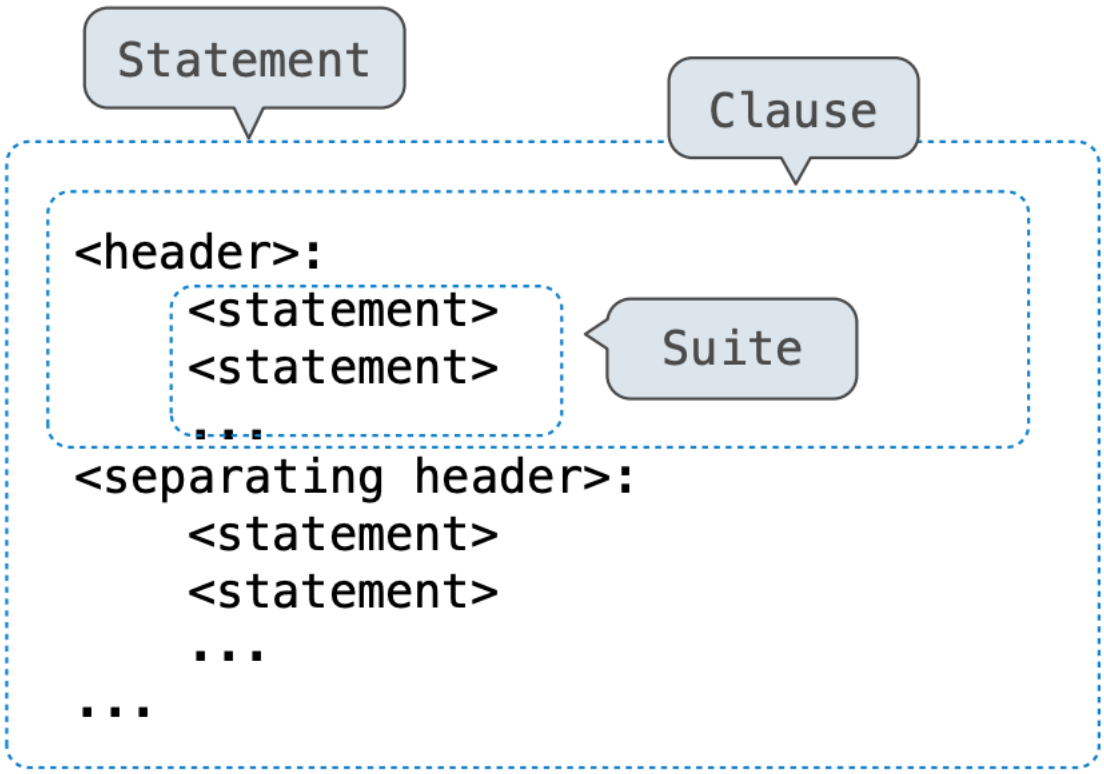
```
>>> <header>:
>>>     <statement>
>>>     <statement>
>>>     ...
>>> <seperating header>
>>>     <statement>
>>>     <statement>
>>>     ...
```

The first header determines a statement's type. The header of a clause controls the suite that follows. A suite is a sequence of statements. To execute a suite means to execute its sequence of statements, in order.
Execution Rule for a sequence of statements:

1. Execute the first statement.
2. Unless directed otherwise, execute the rest.

3.2.2 Conditional Statements

We start with the following example with 1 statement, 3 clauses, 3 headers, and 3 suites.




```
>>> def abs(x):
...     if x < 0:
...             return -x
...     elif x == 0:
...             return 0
...     else:
...             return x
```

Execution Rule for Conditional Statements:
Each clause is considered in order.

1. Evaluate the header's expression.
2. If it is a true value, execute the suite and skip the remaining clauses.

Syntax Tips:

1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause, always at the end.

3.3 Boolean Contexts and Operators

Python supports three boolean operators: **and**, **or**, and **not**.

```
>>> a = 4
>>> a < 2 and a > 0
False
>>> a < 2 or a > 0
True
>>> not (a > 0)
False
```

1. **and** evaluates to **True** only if both operands evaluate to **True**. If at least one operand is **False**, then and evaluates to **False**.
2. **or** evaluates to **True** if at least one operand evaluates to **True**. If both operands are **False**, then or evaluates to **False**.
3. **not** evaluates to **True** if its operand evaluates to **False**. It evaluates to **False** if its operand evalutes to **True**.

Let us look at a more complicated example.

```
>>> (True and (not False)) or ((not True) and False)
```

Before we can solve this, we need to know that boolean operators have an order of operation:
not have the highest priority, then **and**, lastly **or** has the lowest priority. Let us try again.

3.3.1 Short Circuiting

First, let us look at the example below.

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> True or 1/0
True
```

The behavior is curious. The second expression evaluates to **True** because Python's **and** and **or** operators short-circuit. That means they do not necessarily evaluate every operand.
Short-circuiting happens when the operator reaches an operand that allows them to make a conclusion about the expression. For example, **and** will short-circuit as soon as it reaches the first false value because it then knows that not all the values are true.
If **and** and **or** do not short-circuit, they just return the last value. Keep in mind that **and** and **or** don't always return booleans when using values other than **True** and **False**.

Operator	Checks if:	Evaluates from left to right up to:	Example
AND	All values are true	The first false value	`False and 1 / 0` evaluates to `False`
OR	At least one value is true	The first true value	`True or 1 / 0` evaluates to `True`

False values in Python: False, 0, ”, None (more to come)
True values in Python: Anything else.

3.4 Division

True Division: `/` (decimal division)

```
>>> 1 / 5
0.2

>>> 25 / 4
6.25

>>> 4 / 2
2.0

>>> 5 / 0
ZeroDivisionError
```

Floor Division: `//` (integer division)

```
>>> 1 // 5
0

>>> 25 // 4
6

>>> 4 // 2
2

>>> 5 // 0
ZeroDivisionError
```

Modulo: `%` (remainder)

```
>>> 1 % 5
1

>>> 25 % 4
1

>>> 4 % 2
0

>>> 5 % 0
ZeroDivisionError
```

3.5 Exercises

1. ---

```
>>> True and 13
------

>>> False or 0
------

>>> not 10
------

>>> not None
------
>>> True and 1 / 0 and False
------

>>> True or 1 / 0 or False
------
```

```
>>> True and 0
------

>>> False or 1
------

>>> 1 and 3 and 6 and 10 and 15
------

>>> 0 or False or 2 or 1 / 0
------
>>> not 0
------

>>> (1 + 1) and 1
------

>>> 1/0 or True
------

>>> (True or False) and False
------
```

2. Write a function that takes three positive numbers and returns the sum of the squares of the two largest numbers.
Challenge: Use only a single line for the body of the function.

```
def two_of_three(a, b, c):
    """Return x*x + y*y, where x and y are the two largest members of the positive numbers
        a, b, and c.
    >>> two_of_three(1, 2, 3)
    13
    >>> two_of_three(5, 3, 1)
    34
    >>> two_of_three(10, 2, 8)
    164
    >>> two_of_three(5, 5, 5)
    50
    """
    return _____
```

3. Write a function that takes an integer n that is **greater than 1** and returns the largest integer that is smaller than n and evenly divides n.

```
def largest_factor(n):
    """Return the largest factor of n that is smaller than n.

    >>> largest_factor(15) # factors are 1, 3, 5
    5
    >>> largest_factor(80) # factors are 1, 2, 4, 5, 8, 10, 16, 20, 40
    40
    >>> largest_factor(13) # factor is 1 since 13 is prime
```

```
    1
    """
    "*** YOUR CODE HERE ***"
```

Hint: To check if b evenly divides a, you can use the expression a % b == 0, which can be read as, "the remainder of dividing a by b is 0."

4. Douglas Hofstadter's Pulitzer-prize-winning book, Gödel, Escher, Bach, poses the following mathematical puzzle.

   1. Pick a positive integer n as the start.
   2. If n is even, divide it by 2.
   3. If n is odd, multiply it by 3 and add 1.
   4. Continue this process until n is 1.

   The number n will travel up and down but eventually end at 1 (at least for all numbers that have ever been tried – nobody has ever proved that the sequence will terminate). Analogously, a hailstone travels up and down in the atmosphere before eventually landing on earth.

   This sequence of values of n is often called a Hailstone sequence, Write a function that takes a single argument with formal parameter name n, prints out the hailstone sequence starting at n, and returns the number of steps in the sequence.

```
def hailstone(n):
    """Print the hailstone sequence starting at n and return its
    length.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    """
    "*** YOUR CODE HERE ***"
```

5. Write a function that takes in a nonnegative integer and sums its digits. (Using floor division and modulo might be helpful here!)

```
def sum_digits(n):
    """Sum all the digits of n.

    >>> sum_digits(10) # 1 + 0 = 1
    1
    >>> sum_digits(4224) # 4 + 2 + 2 + 4 = 12
    12
```

```
    >>> sum_digits(1234567890)
    45
    """
    "*** YOUR CODE HERE ***"
```

6. Write a function that takes in a number and determines if the digits contain two adjacent 8s.

```
def double_eights(n):
    """Return true if n has two eights in a row.
    >>> double_eights(8)
    False
    >>> double_eights(88)
    True
    >>> double_eights(2882)
    True
    >>> double_eights(880088)
    True
    >>> double_eights(12345)
    False
    >>> double_eights(80808080)
    False
    """
    "*** YOUR CODE HERE ***"
```

7. Let's write a function falling, which is a "falling" factorial that takes two arguments, n and k, and returns the product of k consecutive numbers, starting from n and working downwards.

```
def falling(n, k):
    """Compute the falling factorial of n to depth k.

    >>> falling(6, 3) # 6 * 5 * 4
    120
    >>> falling(4, 0)
    1
    >>> falling(4, 3) # 4 * 3 * 2
    24
    >>> falling(4, 1) # 4
    4
    """
    "*** YOUR CODE HERE ***"
```

4 Higher Order Functions

4.1 Designing Functions

A function's *domain* is the set of all inputs it might possibly take as arguments.
A function's *range* is the set of output values it might possibly return.
A pure function's *behavior* is the relationship it creates between input and output.

4.2 Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame. Look at the following example.

```
def make_adder(n):
"""Return a function that takes one argument k and returns k + n.
>>> add_three = make_adder(3)
>>> add_three(4)
7
"""
   def adder(k):
      return k + n
   return adder
```

Here *make_adder* is a function that returns another function.
The name *add_three* is bound to a function.
adder is a def statement within another def statement.
adder can refer to names in the enclosing function.

4.3 Lambda Expressions

So far, each time we have wanted to define a new function, we needed to give it a name. But for other types of expressions, we do not need to associate intermediate values with a name. That is, we can compute $a * b + c * d$ without having to name the subexpressions $a * b$ or $c * d$, or the full expression. In Python, we can create function values on the fly using lambda expressions, which evaluate to unnamed functions. A lambda expression evaluates to a function that has a single return expression as its body. Assignment and control statements are not allowed. Lambda functions are not common in Python, but they are important in general.

We can understand the structure of a lambda expression by constructing a corresponding English sentence:

$$\underbrace{\text{lambda}}_{\text{A function that}} \quad \underbrace{x}_{\text{takes } x} \quad \underbrace{:}_{\text{and returns}} \quad \underbrace{f(g(x))}_{f(g(x))}$$

It is important that lambda functions do not have "return" keywords and it must return a single expression.

```
>>> square = lambda x: x * x
>>> square(4)
16
```

Lambda Expressions Versus Def Statements.

```
>>> square = lambda x: x * x
>>> def square(x):
      return x * x
```

1. Both create a function with the same domain, range, and behavior.
2. Both bind that function to the name square.
3. Only the def statement gives the function an intrinsic name, which shows up inenvironment diagrams but doesn't affect execution (unless the function is printed).

4.4 Exercises

1.

```
>>> lambda x: x # A lambda expression with one parameter x
------

>>> a = lambda x: x # Assigning the lambda function to the name a
>>> a(5)
------

>>> (lambda: 3)() # Using a lambda expression as an operator in a call exp.
------

>>> b = lambda x: lambda: x # Lambdas can return other lambdas!
>>> c = b(88)
>>> c
------

>>> c()
------

>>> d = lambda f: f(4) # They can have functions as arguments as well.
>>> def square(x):
...     return x * x
>>> d(square)
------
```

2.
```
>>> z = 3
>>> e = lambda x: lambda y: lambda: x + y + z
>>> e(0)(1)()
------

>>> f = lambda z: x + z
>>> f(3)
------
```

3.
```
>>> higher_order_lambda = lambda f: lambda x: f(x)
>>> g = lambda x: x * x
>>> higher_order_lambda(2)(g) # Which argument belongs to which function call?
------

>>> higher_order_lambda(g)(2)
------

>>> call_thrice = lambda f: lambda x: f(f(f(x)))
>>> call_thrice(lambda y: y + 1)(0)
------

>>> print_lambda = lambda z: print(z) # When is the return expression of a lambda
    expression executed?
>>> print_lambda
------

>>> one_thousand = print_lambda(1000)
------

>>> one_thousand
------
```

4.
```
>>> def even(f):
...     def odd(x):
...         if x < 0:
...             return f(-x)
...         return f(x)
...     return odd
>>> steven = lambda x: x
>>> stewart = even(steven)
>>> stewart
------

>>> stewart(61)
------

>>> stewart(-4)
------
```

5.

```
>>> def cake():
...     print('beets')
...     def pie():
...         print('sweets')
...         return 'cake'
...     return pie
>>> chocolate = cake()
------

>>> chocolate
------

>>> chocolate()
------

>>> more_chocolate, more_cake = chocolate(), cake
------

>>> more_chocolate
------

>>> def snake(x, y):
...     if cake == more_cake:
...         return lambda y: x + y
...     else:
...         return x + y
>>> snake(10, 20)
------

>>> snake(10, 20)(30)
------

>>> cake = 'cake'
>>> snake(10, 20)
------
```

6. Lambdas and Currying: We can transform multiple-argument functions into a chain of single-argument, higher order functions by taking advantage of lambda expressions. This is useful when dealing with functions that take only single-argument functions. Write a function *lambda_curry2* that will curry any two argument function using lambdas.

```
def lambda_curry2(func):
    """
    Returns a Curried version of a two-argument function FUNC.
    >>> from operator import add
    >>> curried_add = lambda_curry2(add)
    >>> add_three = curried_add(3)
    >>> add_three(5)
    8
    """
    "*** YOUR CODE HERE ***"
    return ______
```

7. Make Adder with a Lambda: Implement the *make_adder* function below using a single return statement that returns the value of a lambda expression.

```
def make_adder(n):
    """Return a function that takes an argument K and returns N + K.

    >>> add_three = make_adder(3)
    >>> add_three(1) + add_three(2)
    9
    >>> make_adder(1)(2)
    3
    """
    "*** YOUR CODE HERE ***"
    return 'REPLACE ME'
```

8. Composite Identity Function: Write a function that takes in two single-argument functions, f and g, and returns another function that has a single parameter x. The returned function should return True if $f(g(x))$ is equal to $g(f(x))$. You can assume the output of $g(x)$ is a valid input for f and vice versa. You may use the *compose1* function defined below.

```
def compose1(f, g):
    """Return the composition function which given x, computes f(g(x)).

    >>> add_one = lambda x: x + 1     # adds one to x
    >>> square = lambda x: x**2
    >>> a1 = compose1(square, add_one) # (x + 1)^2
    >>> a1(4)
    25
    >>> mul_three = lambda x: x * 3  # multiplies 3 to x
    >>> a2 = compose1(mul_three, a1) # ((x + 1)^2) * 3
    >>> a2(4)
    75
    >>> a2(5)
    108
    """
    return lambda x: f(g(x))

def composite_identity(f, g):
    """
    Return a function with one parameter x that returns True if f(g(x)) is
    equal to g(f(x)). You can assume the result of g(x) is a valid input for f
    and vice versa.

    >>> add_one = lambda x: x + 1     # adds one to x
    >>> square = lambda x: x**2
    >>> b1 = composite_identity(square, add_one)
    >>> b1(0)                         # (0 + 1)^2 == 0^2 + 1
    True
    >>> b1(4)                         # (4 + 1)^2 != 4^2 + 1
    False
    """
    "*** YOUR CODE HERE ***"
```

9. Define a function cycle that takes in three functions f_1, f_2, f_3, as arguments. cycle will return another function that should take in an integer argument n and return another function. That final function should take in an argument x and cycle through applying f_1, f_2, and f_3 to x, depending on what n was. Here's what the final function should do to x for a few values of n:
$n = 0$, return x
$n = 1$, apply f_1 to x, or return $f_1(x)$
$n = 2$, apply f_1 to x and then f_2 to the result of that, or return $f_2(f_1(x))$
$n = 3$, apply f_1 to x, f_2 to the result of applying f_1, and then f_3 to the result of applying f_2, or $f_3(f_2(f_1(x)))$
$n = 4$, start the cycle again applying f_1, then f_2, then f_3, then f_1 again, or $f_1(f_3(f_2(f_1(x))))$
And so forth.
Hint: most of the work goes inside the most nested function.

```
def cycle(f1, f2, f3):
    """Returns a function that is itself a higher-order function.

    >>> def add1(x):
    ...     return x + 1
    >>> def times2(x):
    ...     return x * 2
    >>> def add3(x):
    ...     return x + 3
    >>> my_cycle = cycle(add1, times2, add3)
    >>> identity = my_cycle(0)
    >>> identity(5)
    5
    >>> add_one_then_double = my_cycle(2)
    >>> add_one_then_double(1)
    4
    >>> do_all_functions = my_cycle(3)
    >>> do_all_functions(2)
    9
    >>> do_more_than_a_cycle = my_cycle(4)
    >>> do_more_than_a_cycle(2)
    10
    >>> do_two_cycles = my_cycle(6)
    >>> do_two_cycles(1)
    19
    """
    "*** YOUR CODE HERE ***"
```