EXPRESSIONS, STATEMENTS, FUNCTIONS, CONTROL AND HIGHER ORDER FUNCTIONS

CS 7 NOTE 1

Compiled by Eric Khumalo for Emzini weCode

Expressions

Expressions describe a computation and evaluate to a value.

Primitive Expressions

A ***primitive expression*** is a single evaluation step: you either look up the value of a name, or take the literal value. For example, numbers, variable names, and strings are all primitive expressions.

```
>>> 73
73
>>> 'Welcome to CS 7'
'Welcome to CS 7'
```

Call Expressions

Call expressions are expressions that involve a call to some function. Call expressions are just another type of expression, called a ***compound expression***. A call expression invokes a function, which may or may not accept arguments, and returns the function's return value. Recall the syntax of a function call:

$$\underbrace{\text{add}}_{\text{Operator}}\ (\ \underbrace{\ 2\ ,}_{\text{Operand 0}}\ \underbrace{\ 3\ \ }_{\text{Operand 1}})$$

Every call expression is required to have a set of parentheses delimiting its comma-separated operands. To evaluate a function call:

1. First evaluate the operator, and then the operands (from left to right).
2. Apply the function (the value of the operator) to the arguments (the values of the operands).

If the operands are nested function calls, apply the two steps to the operands.

Statements

A statement in Python is executed by the interpreter to achieve an effect.

For example, we can talk about an assignment statement. In an assignment statement, we ask Python to assign a certain value to a variable name.
We can assign values to names using the = operator
Every assignment will assign the value on the right of the = operator to the name on the left.
For example:

```
>>> foo = 7
```

Binds the value 7 to the name foo.
We can also bind values to names using the import statement.

```
>>> from operator import add
```

Will bind the name add to the add function.

Function definitions also bind the name of the function to that function.

```
>>> def foo():
...     return 0
```

Will bind foo the function defined above

Assignments are also not permanent, if two values are bound to the same name only the assignment executed last will remain.

```
>>> add = 73
>>> from operator import add
```

After the above two statements are executed the value `73` will no longer be bound to add but rather the add function from operator will.

Pure and Non-Pure Functions

Pure function — It only produces a return value (no side effects), and always evaluates to the same result, given the same argument value(s).
Non-Pure function — It produces side effects, such as printing to the screen.

Control

Control refers to directing the computer to selectively choose which lines of code get executed

Acknowledgements: These notes are an adaptation from CS61A material at UC Berkeley. .

Conditional Statements

Conditional statements allow programs to execute different lines of code depending on the current state. A typical if-else set of statements will have the following structure:

```
if : <conditional expression>:
    <suite of statements>
elif <conditional expression>:
    <suite of statements>
else:
    <suite of statements>
```

Recall the following points:

- The `else` and `elif` clauses are optional, and you can have any number of `elif` clauses.
- A **conditional expression** is an expression that evaluates to either a true value (`True`, a non-zero integer, etc.) or a false value (`False`, `0`, `None`, etc.).
- Only the **suite** that is indented under the first `if/elif` that has a **conditional expression** that evaluates to `True` will be executed.
- If none of the **conditional expression**s are `True`, then the `else` suite is executed. There can only be one `else` clause in a conditional statement!

```
>>> if 2 + 3:
 ... print(6)
6
```

Here's some example code

```
>>> def mystery(x):
...     if x > 0:
...        print(x)
...     else:
...        x(mystery)
...
>>> mystery(5)
5
>>> mystery(-1)
TypeError: ’int’ object is not callable
```

Boolean Operators

Python also includes the **boolean operators** `and`, `or`, and `not`. These operators are used to combine and manipulate boolean values.

- `not` returns the opposite truth value of the following expression.
- `and` short-circuits at the first `False` value and returns it. If all values evaluate to `True`, the last value is returned.

- `or` short-circuits at the first `True` value and returns it. If all values evaluate to `False`, the last value is returned.

```
>>> not None
True
>>> not True
False
>>> -1 and 0 and 1
0 >>> False or 9999 or 1/0
9999
```

Iteration

Using conditional statements we can ignore statements. On the other hand using iteration we can repeat statements multiple times. A common iterative block of code is the while statement. The structure is as follows:

```
while <conditional clause>:
    <body of statements>
```

This block of code literally means while the conditional clause evaluates to `True`, execute the body of statements over and over.

```
>>> def countdown(x):
...     while x > 0:
...         print(x)
...         x = x - 1
...     print("Blastoff!")
...
>>> countdown(3)
3
2
1
Blastoff!
```

Functions

A higher order function (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both.

Functions as Argument Values

Acknowledgements: These notes are an adaptation from CS61A material at UC Berkeley. .

Suppose we would like to square or double every natural number from 1 to `n` and print the result as we go. Using the functions `square` and `double`, each of which are functions that take one argument that do as their name implies, we can define the functions as follows:

```
def square_every_number(n):
    i = 1
    while i <= n:
        print(square(i))
        i = i + 1


def double_every_number(n):
    i = 1
    while i <= n:
        print(double(i))
        i = i + 1
```

Note that the only thing different about `square_every_number` and `double_every_number` is just what function we call on `n` when we print it. Wouldn't it be nice to generalize functions of this form into something more convenient? When we pass in the number, couldn't we specify, also, what we want to do to each number < `n.`
To do that, we can define a higher order function called `every`. `every` takes in the function you want to apply to each element as an argument, and applies it to n natural numbers starting from 1. So to write `square_every_number`, we can simply do:

```
def square_every_number(n):
    every(square, n)
```

Equivalently, to write double every number, we can write:

```
def double_every_number(n):
    every(double, n)
```

Note: These functions are not pure — as defined below, every will actually print values to the screen.

We can now implement the function `every` that takes in a function `func` and a number `n`, and applies that function to the first n numbers from 1 and prints the result along the way

```
def every(func, n):
    i = 1
    while i <= n:
        print(func(i))
```

Acknowledgements: These notes are an adaptation from CS61A material at UC Berkeley. .

```
        i = i + 1
```

Functions as Return Values
Often, we will need to write a function that returns another function. One way to do this is to define a function inside another function:

```
def outer(x):
    def inner(y):
        ...
    return inner
```

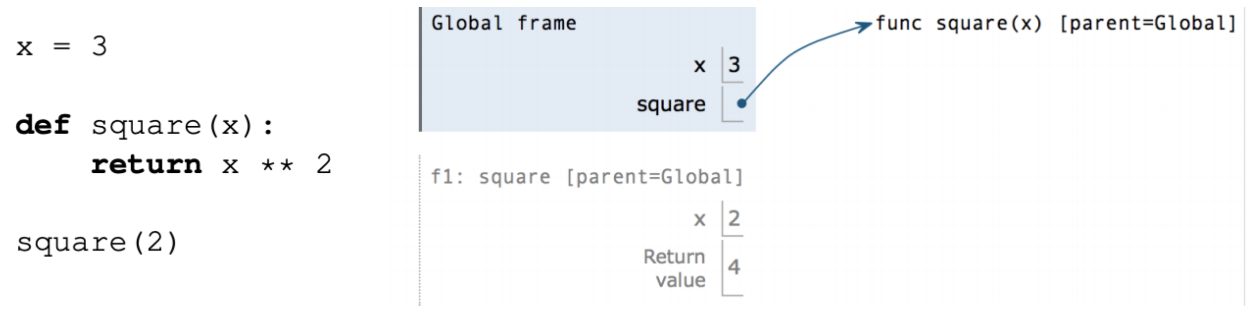
The return value of `outer` is the function `inner`. This is a case of a function returning a function. In this example, `inner` is defined inside of `outer.` Although this is a common pattern, we can also define `inner` outside of `outer` and still use the same `return` statement.

```
def inner(y):
    ...

def outer(x):
    return inner
```

Environment Diagrams

An **environment diagram** keeps track of all the variables that have been defined and the values they are bound to.



```
x = 3

def square(x):
    return x ** 2

square(2)
```

When Python executes ***assignment statements*** (like `x = 3`), it records the variable name and the value:

1. Evaluate the expression on the right side of the = sign
2. Write the variable name and the expression's value in the current frame.

When Python executes `def` *statements*, it records the function name and binds it to a function object:

1. Write the function name (`square`) in the frame and point it to a function object (`func square(x) [parent=Global]`). The `[parent=Global]` denotes the frame in which the function was `defined.`

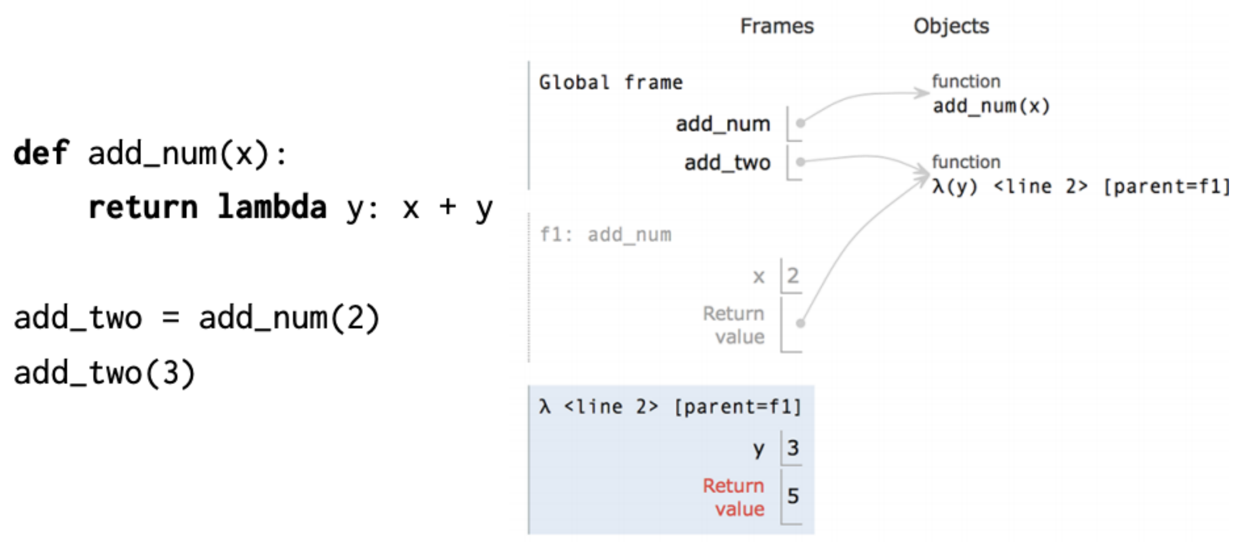
When Python executes a ***call expression*** (like `square(2)`), it creates a new frame to keep track of local variables.

- Draw a new frame *. Label it with
    - a unique index (`f1`, `f2`, `f3` and so on)
    - the intrinsic name of the function (`square`)
    - the parent frame (`[parent=Global]`)
- Bind the formal parameters to the arguments passed in (e.g. bind `x` to 3).
- Evaluate the body of the function.

The **intrinsic name** is the name in the function object. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.
If a function does not have a return value, it implicitly returns `None`. Thus, the "Return value" box should contain `None`.
* Since we do not know how built-in functions like `add(...)` or `min(...)` are implemented, we do not draw a new frame when we call built-in functions.

Higher Order Functions in Environment Diagrams

An environment diagram keeps track of all the variables that have been defined and the values they are bound to. However, values are not necessarily only integers and strings. Environment diagrams can model more complex programs that utilize higher order functions.



```python
def add_num(x):
    return lambda y: x + y

add_two = add_num(2)
add_two(3)
```

Acknowledgements: These notes are an adaptation from CS61A material at UC Berkeley. .

Lambdas are represented similarly to functions in environment diagrams, but since they lack intrinsic names, the lambda symbol **(λ)** is used instead. The parent of any function (including lambdas) is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call **`add_two`** (which is really the lambda function), we need to know what `x` is in order to compute `x + y`. Since `x` is not in the frame `f2,` we look at the frame's parent, which is `f1`. There, we find `x` is bound to `2`.

As illustrated above, higher order functions that return a function have their return value represented with a pointer to the function object.

Lambda Expressions

A lambda expression evaluates to a function, called a lambda function.
In the code above, `lambda y: x + y` is a lambda expression, and can be read as a function that takes in one parameter `y` and returns `x + y`

A lambda expression by itself evaluates to a function but does not bind it to a name. Also note that the return expression of this function is not evaluated until the lambda is called. This is similar to how defining a new function using a def statement does not execute the functions body until it is later called.

```
>>> what = lambda x : x + 5
>>> what
<function <lambda> at 0xf3f490>
```

Unlike def statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions.

```
>>> (lambda y: y + 5)(4)
9
>>> (lambda f, x: f(x))(lambda y: y + 1, 10)
11
```

Challenge Yourself!

Environment Diagrams

Now use all the knowledge from the information above and the previous notes to draw the environment diagram that results from executing the code below.

```
def curry2(h):
    def f(x):
```

Acknowledgements: These notes are an adaptation from CS61A material at UC Berkeley. .

```
        def g(y):
            return h(x, y)
        return g
    return f
make_adder = curry2(lambda x, y: x + y)
add_three = make_adder(3)
add_four = make_adder(4)
five = add_three(2)
```

Try challenging yourself by writing `curry2` as a lambda function

Writing Higher Order Functions

Write a function that takes in a function `cond` and a number `n` and prints numbers from `1` to `n` where calling `cond` on that number returns `True`.

```
def keep_ints(cond, n):
      """Print out all integers 1..i..n where cond(i) is true

      >>> def is_even(x):
      ...   # Even numbers have remainder 0 when divided by 2.
      …     return x % 2 == 0
      >>> keep_ints(is_even, 5)
      2
      4
      """
      ***YOUR CODE HERE***
```

Acknowledgements: These notes are an adaptation from CS61A material at UC Berkeley. .