

EXPRESSIONS, STATEMENTS, FUNCTIONS, CONTROL AND HIGHER ORDER FUNCTIONS

CS 7 NOTE 1

Compiled by Eric Khumalo

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:

add	(2	,	3)
<u> </u>		<u> </u>		<u> </u>	
Operator		Operand 0		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

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 epression>:
```

```
    <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** clause.
- 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 expressions** 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

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))
        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
```

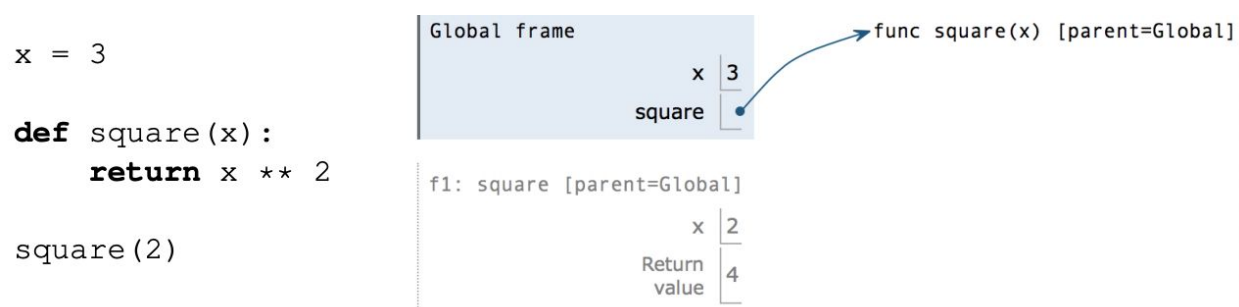
The return value of `outer` is the function `inner`. This is a case of a function returning a function. In this example, `inner` is defined inside of `outer`. Although this is a common pattern, we can also define `inner` outside of `outer` and still use the same `return` statement.

```
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.



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.