CS 61A
Summer 2018

Control and Environments

Discussion 1: June 21, 2018

1 Control

Control structures direct the flow of logic in a program. For example, conditionals (`if-elif-else`) allow a program to skip sections of code, while iteration (`while`), allows a program to repeat a section.

If statements

Conditional statements let programs execute different lines of code depending on certain conditions. Let's review the `if- elif-else` syntax.

```
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 a 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` with a **conditional expression** evaluating to a true value will be executed.
- If none of the **conditional expressions** evaluate to a true value, then the `else` suite is executed. There can only be one `else` clause in a conditional statement!

Boolean Operators

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

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

- `not` returns the opposite truth value of the following expression.
- `and` stops evaluating any more expressions (short-circuits) once it reaches the first false value and returns it. If all values evaluate to a true value, the last value is returned.
- `or` short-circuits at the first true value and returns it. If all values evaluate to a false value, the last value is returned.

Questions

1.1 Alfonso will only wear a jacket outside if it is below 60 degrees or it is raining.

Write a function that takes in the current temperature and a boolean value telling if it is raining and returns `True` if Alfonso will wear a jacket and `False` otherwise.

First, try solving this problem using an if statement.

```
def wears_jacket_with_if(temp, raining):
    """
    >>> wears_jacket(90, False)
    False
    >>> wears_jacket(40, False)
    True
    >>> wears_jacket(100, True)
    True
    """
```

Note that we'll either return `True` or `False` based on a single condition, whose truthiness value will also be either `True` or `False`. Knowing this, try to write this function using a single line.

```
def wears_jacket(temp, raining):
```

While loops

To repeat the same statements multiple times in a program, we can use iteration. In Python, one way we can do this is with a **while loop**.

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

As long as `<conditional clause>` evaluates to a true value, `<body of statements>` will continue to be executed. The conditional clause gets evaluated each time the body finishes executing.

Questions

1.2 Write a function that returns `True` if `n` is a prime number and `False` otherwise. After you have a working solution, think about potential ways to make your solution more *efficient*.

Hint: use the `%` operator: `x % y` returns the remainder of `x` when divided by `y`.

```
def is_prime(n):
    """
    >>> is_prime(10)
    False
    >>> is_prime(7)
    True
    """
```

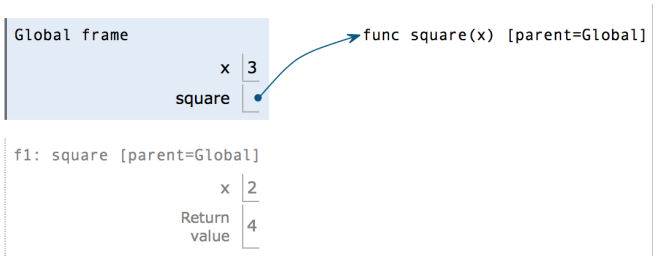
2 Environment Diagrams

An **environment diagram** keeps track of all the variables that have been defined and the values they are bound to. We will be using this tool throughout the course to understand complex programs involving several different objects and function calls.

```
x = 3

def square(x):
    return x ** 2

square(2)
```



Remember that programs are simply a set of statements, or instructions, so drawing diagrams that represent these programs also involve following sets of instructions! Let's dive in.

Assignment Statements

Assignment statements, such as `x = 3`, define variables in programs. To execute one in an environment diagram, record 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.

2.1 Use these rules to draw a simple diagram for the assignment statements below.

```
x = 10 % 4
y = x
x **= 2
```

def Statements

`def` *statements* create function objects and bind them to a name. To diagram `def` statements, record the function name and bind the function object to the name. It's also important to write the **parent frame** of the function, which is where the function is defined.

1. Draw the function object to the right-hand-side of the frames, denoting the intrinsic name of the function, its parameters, and the parent frame (e.g. `func square(x) [parent = Global]`.
2. Write the function name in the current frame and draw an arrow from the name to the function object.

2.2 Use these rules and the rules for assignment statements to draw a diagram for the code below.

```
def double(x):
    return x * 2

def triple(x):
    return x * 3

hmmm = double
double = triple
```

Call Expressions

Call expressions, such as `square(2)`, apply functions to arguments. When executing call expressions, we create a new frame in our diagram to keep track of local variables:

1. Evaluate the operator, which should evaluate to a function.
2. Evaluate the operands from left to right.
3. Draw a new frame, labelling it with the following: [1]
   - A unique index (`f1`, `f2`, `f3`, ...)
   - The **intrinsic name** of the function, which is the name of the function object itself. For example, if the function object is `func square(x) [parent=Global]`, the intrinsic name is `square`.
   - The parent frame (`[parent=Global]`)
4. Bind the formal parameters to the argument values obtained in step 2 (e.g. bind `x` to 3).
5. Evaluate the body of the function in this new frame until a return value is obtained. Write down the return value in the frame.

If a function does not have a return value, it implicitly returns `None`. In that case, the "Return value" box should contain `None`.

2.3 Let's put it all together! Draw an environment diagram for the following code.

```
def double(x):
    return x * 2

hmmm = double
wow = double(3)
hmmm(wow)
```

[1] 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.

3 Higher Order 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 Arguments

One way a higher order function can manipulate other functions is by taking functions as input (an argument). Consider this higher order function called `negate`.

```
def negate(f, x):
    return -f(x)
```

`negate` takes in a function `f` and a number `x`. It doesn't care what exactly `f` does, as long as `f` is a function, takes in a number and returns a number. Its job is simple: call `f` on `x` and return the negation of that value.

Questions

3.1 Here are some possible functions that can be passed through as `f`.

```
def square(n):
    return n * n

def double(n):
    return 2 * n
```

What will the following Python statements display?

```
>>> negate(square, 5)


>>> negate(double, -19)


>>> negate(double, negate(square, -4))
```

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 of a 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. However, note that in this second example (unlike the first example), `inner` doesn't have access to variables defined within the `outer` function, like `x`.

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

Questions

3.1 Draw an environment diagram for the following program.

```
def outer(n):
    def inner(m):
        return n - m
    return inner

outer(61)
f = outer(10)
f(4)
```

3.2 Using the same definition of `outer` above, what is the return value of `outer(5)(4)`?

Extra Questions

3.3 Draw the environment diagram that results from executing the code below.

```
from operator import add

six = 1

def ty(one, a):
    summer = one(a, six)
    return summer

six = ty(add, 6)
summer = ty(add, 6)
```