CS 61A
Summer 2017

Control and Environments

Discussion 1: June 22, 2017

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. Fill in the function `wears_jacket` which 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.

This should only take one line of code!

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

1.2 To handle discussion section overflow, TAs may direct students to a more empty section that is happening at the same time. Write the function `handle_overflow`, which takes in the number of students at two sections and prints out what to do if either section exceeds 30 students. **Note:** Don't worry about printing "spot" for singular values and "spots" for multiple values.

```
def handle_overflow(s1, s2):
    """
    >>> handle_overflow(27, 15)
    No overflow.
    >>> handle_overflow(35, 29)
    1 spot left in Section 2.
    >>> handle_overflow(20, 32)
    10 spots left in Section 1.
    >>> handle_overflow(35, 30)
    No space left in either section.
    """
```

While loops

Iteration lets a program repeat statements multiple times. A common iterative block of code is the **`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

1.1 What is the result of evaluating the following code?

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

def so_slow(num):
    x = num
    while x > 0:
        x = x + 1
    return x / 0

square(so_slow(5))
```

1.2 Fill in the `is_prime` function, which 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):
```

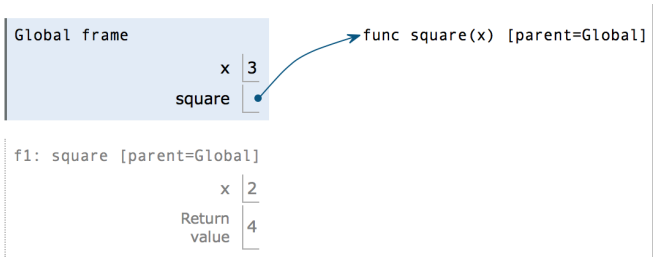
2 Environment Diagrams

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 you execute *assignment statements* in an environment diagram (like `x = 3`), you need to 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.

When you execute `def` *statements*, you need to record the function name and bind the function object to the name.

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

When you execute a *call expression* (like `square(2)`), you need to create a new frame to keep track of local variables.

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

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

[a] 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.

Questions

2.1 Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

2.2 Draw the environment diagram so we can visualize exactly how Python evaluates the code. What is the output of running this code in the interpreter?

```
>>> from operator import add
>>> def sub(a, b):
...     sub = add
...     return a - b
>>> add = sub
>>> sub = min
>>> print(add(2, sub(2, 3)))
```