CS 61A
Summer 2020

# Control and Environments

Discussion 1: June 25, 2020

## 1 Control

**Control structures** direct the flow of a program using logical statements. For example, conditionals (`if-elif-else`) allow a program to skip sections of code, and 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 an expression that evaluates to either a truthy value (`True`, a non-zero integer, etc.) or a falsy value (`False`, `0`, `None`, `""`, `[]`, etc.).
- Only the **suite** that is indented under the first `if`/`elif` whose **conditional expression** evaluates 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` returns the opposite truth value of the following expression (so `not` will always return either `True` or `False`).
- `and` evaluates expressions in order and stops evaluating (short-circuits) once it reaches the first false value, and then 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.

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

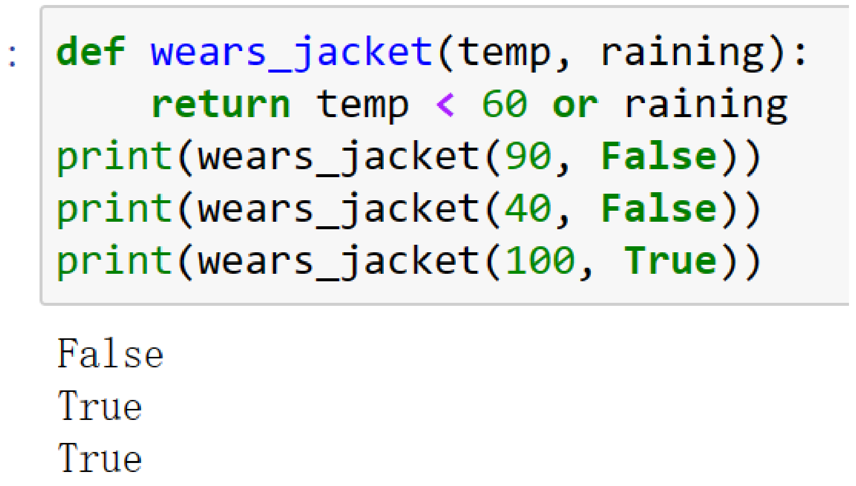
## Questions

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_with_if(90, False)
    False
    >>> wears_jacket_with_if(40, False)
    True
    >>> wears_jacket_with_if(100, True)
    True
    """
```



```
:
def wears_jacket(temp, raining):
    return temp < 60 or raining
print(wears_jacket(90, False))
print(wears_jacket(40, False))
print(wears_jacket(100, True))
```

```
False
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):
```

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*

## 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.

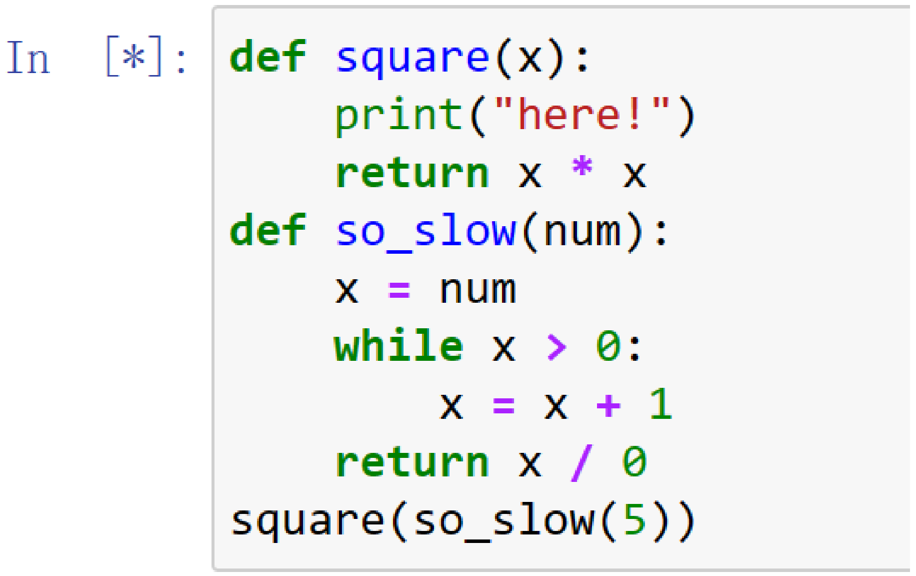
## Questions

1.2 What is the result of evaluating the f

```
def square(x):
    print("here!")
    return x * x

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

square(so_slow(5))
```



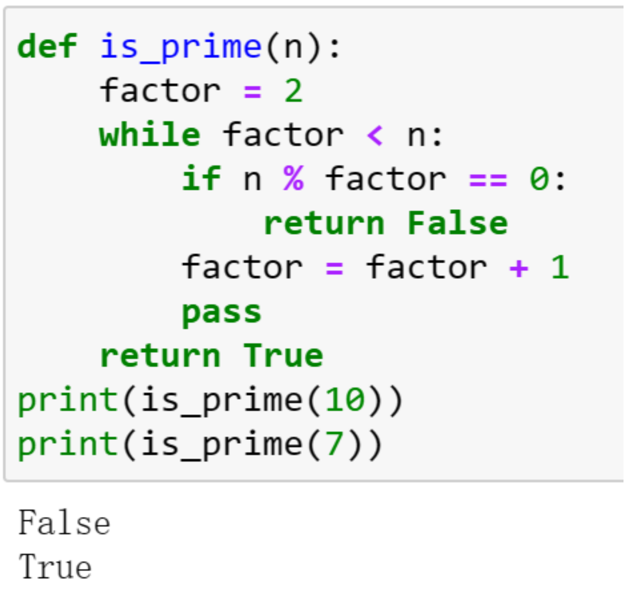
```
In [*]: def square(x):
            print("here!")
            return x * x
        def so_slow(num):
            x = num
            while x > 0:
                x = x + 1
            return x / 0
        square(so_slow(5))
```

1.3 Write a function that returns `True` if a positive integer `n` is a prime number and `False` otherwise.

A prime number $n$ is a number that is not divisible by any numbers other than 1 and $n$ itself. For example, 13 is prime, since it is only divisible by 1 and 13, but 14 is not, since it is divisible by 1, 2, 7, and 14.

**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
    """
```



```
def is_prime(n):
    factor = 2
    while factor < n:
        if n % factor == 0:
            return False
        factor = factor + 1
        pass
    return True
print(is_prime(10))
print(is_prime(7))
```

```
False
True
```

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*

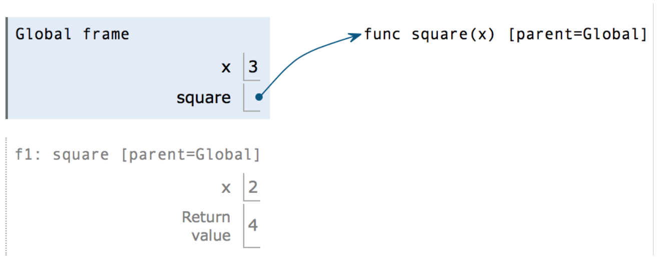
# 2 Environment Diagrams

An **environment diagram** is a model we use to keep 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 assignments 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 involves 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
```

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*

## 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. **Very important note:** Assignments for `def` statements use pointers to functions, which can have different behavior than primitive assignments.

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]`. [1]
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

hat = double
double = triple
```

[1]When importing functions, we still create a function object in the environment diagram, bound to the name of the imported function. However, the parent and parameters of an imported function is unknown so only the function's name is included. For example, if we imported the function `add`, the function object would just be `add(...)`

## 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: [2]
   - 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)
```

[2] Since we do not know how built-in functions like `min(...)` or imported functions like `add(...)` are implemented, we do *not* draw a new frame when we call them, for our own sakes.

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

```
def f(x):
        return x

def g(x, y):
        if x(y):
                return not y
        return y

x = 3
x = g(f, x)
f = g(f, 0)
```

1. **Proceed with call-tion**
For each of the expressions in the table 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 value is displayed, write "Function". The first row has been provided as an example. Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is None. Assume that you have started python3 and executed the following statements. Changes to values persist across subproblems.

```
x = 3

def p(rint):
        print(rint)

def g(x, y):
        if x:
                print("one")
        elif x:
                print(True, x) # Does x being truth-y affect the printed value?
        if y:
                print(True, y) # Does y being truth-y affect the printed value?
        else:
                print(False, y) # Does y being false-y affect the printed value?
        return print(p(y)) + x
```

| **Expression** | **Interactive Output** |
|---|---|
| `print(4, 5) + 1` | 4 5<br>Error |
| `2 * 2 * 1 + x * x` | 13 |
| `print(3 * 3 * 1)` | 9 |
| `print(x + 1 * x + 1)` | 7 |
| `print(print(x + 1 * x + 1))` | 7<br>None |
| `print(print(x + 1 * x + 1) + 1)` | 7<br>Error |
| `print(p("rint"))` | rint<br>None |
| `x, y = 2, x`<br>`g(y, x)` | one<br>True 2<br>2<br>None<br>Error |
| `g(y, p("rint"))` | rint<br>one<br>False None<br>None<br>None<br>Error |