CS 61A
Summer 2023

Functions, Control, Environment Diagrams

Discussion 1: June 22, 2023

Control Structures

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.

Conditional Statements

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

- The `elif` and `else` 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 first `if`/`elif` expression that evaluates to a **truthy** value will have its corresponding indented suite 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.

Here's the general form:

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

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 boolean value of the following expression, and will always return either `True` or `False`.
- `and` evaluates expressions in order and stops evaluating (short-circuits) once it reaches the first falsy value, and then returns it. If all values evaluate to a truthy value, the last value is returned.
- `or` evalutes expressions in order and short-circuits at the first truthy value and returns it. If all values evaluate to a falsy value, the last value is returned.

For example:

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

Q1: Case Conundrum

In this question, we will explore the difference between `if` and `elif`.

What is the result of evaluating the following code?

```
def special_case():
    x = 10
    if x > 0:
        x += 2
    elif x < 13:
        x += 3
    elif x % 2 == 1:
        x += 4
    return x

special_case()
```

What is the result of evaluating this piece of code?

```
def just_in_case():
    x = 10
    if x > 0:
        x += 2
    if x < 13:
        x += 3
    if x % 2 == 1:
        x += 4
    return x

just_in_case()
```

How about this piece of code?

```
def case_in_point():
    x = 10
    if x > 0:
        return x + 2
    if x < 13:
        return x + 3
    if x % 2 == 1:
        return x + 4
    return x

case_in_point()
```

Which of these code snippets result in the same output, and why? Based on your findings, when do you think using a series of `if` statements has the same effect as using both `if` and `elif` cases?

The calls to `special_case` and `case_in_point` both return 12, while the call to `just_in_case` returns 19. Since the number 10 satisfies all three conditions in each function, the value of the variable `x` is incremented three times when `just_in_case` is called. A series of `if` statements has the same effect as using both `if` and `elif` cases if each `if` clause ends in a `return` statement.

Q2: Jacket Weather?

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. This function should return `True` if Alfonso will wear a jacket and `False` otherwise.

Try solving this problem using an `if` statement.

> **Note:** Since 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_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
    """
    if temp < 60 or raining:
        return True
    else:
        return False

#Alternate Solution
def wears_jacket_alt(temp, raining):
    return temp < 60 or raining
```

Q3: Nearest Ten

Write a function that takes in a positive number n and rounds n to the nearest ten.

Solve this problem using an `if` statement.

```
def nearest_ten(n):
    """
    >>> nearest_ten(0)
    0
    >>> nearest_ten(4)
    0
    >>> nearest_ten(5)
    10
    >>> nearest_ten(61)
    60
    >>> nearest_ten(2023)
    2020
    """
    if n % 10 < 5:
        return n // 10 * 10
    else:
        return (n // 10 + 1) * 10

#Alternate Solution
def nearest_ten(n):
    return (n + 5) // 10 * 10
```

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>:
    <statements body>
```

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

Q4: Square So Slow

What is the result of evaluating the following code?

```
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))
```

Hint: What happens to `x` over time?

Solution: This program results in an infinite loop because `x` will always be greater than 0; `x / 0` is never executed. We also know that `here!` is never printed since the operand `so_slow(5)` must be evaluated before `function square (x)` can be called.

Here's a video walkthrough.

Q5: Is Prime?

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
    >>> is_prime(1) # one is not a prime number!!
    False
    """
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

Q6: Fizzbuzz

Implement the fizzbuzz sequence, which prints out a *single statement* for each number from 1 to `n`. For a number `i`,

- If `i` is divisible by both 3 and 5, then we print "fizzbuzz".
- If `i` is divisible by 3 only, then we print "fizz".
- If `i` is divisible by 5 only, then we print "buzz".
- Otherwise, we print the number `i` by itself.

Implement `fizzbuzz(n)` here:

```python
def fizzbuzz(n):
    """
    >>> result = fizzbuzz(16)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    16
    >>> result is None  # No return value
    True
    """
    i = 1
    while i <= n:
        if i % 3 == 0 and i % 5 == 0:
            print('fizzbuzz')
        elif i % 3 == 0:
            print('fizz')
        elif i % 5 == 0:
            print('buzz')
        else:
            print(i)
        i += 1
```

To print something for each number from 1 to `n`, we can use a loop that goes through each number, and then check which of the cases applies using `if-elif-else` to figure out what to print.

Students should be careful about the order in which they have their `if-elif` statements: we want to first check if `i` is divisible by both 3 and 5, or otherwise we will end up printing "fizz" if the student checked for divisibility by 3 first (or "buzz" if the student checked for divisibility by 5 first) rather than "fizzbuzz".

Video walkthrough

Q7: Unique Digits

Write a function that returns the number of unique digits in a positive integer.

> **Hints:** You can use `//` and `%` to separate a positive integer into its one's digit and the rest of its digits.
>
> You may find it helpful to first define a function `has_digit(n, k)`, which determines whether a number `n` has digit `k`.

```
def unique_digits(n):
    """Return the number of unique digits in positive integer n.

    >>> unique_digits(8675309) # All are unique
    7
    >>> unique_digits(13173131) # 1, 3, and 7
    3
    >>> unique_digits(101) # 0 and 1
    2
    """
    unique = 0
    while n > 0:
        last = n % 10
        n = n // 10
        if not has_digit(n, last):
            unique += 1
    return unique

# Alternate solution
def unique_digits_alt(n):
    unique = 0
    i = 0
    while i < 10:
        if has_digit(n, i):
            unique += 1
        i += 1
    return unique

def has_digit(n, k):
    """Returns whether K is a digit in N.
    >>> has_digit(10, 1)
    True
    >>> has_digit(12, 7)
    False
    """
    while n > 0:
        last = n % 10
        n = n // 10
        if last == k:
            return True
    return False
```

We have provided two solutions: - In one solution, we look at the current digit, and check if the rest of the number contains that digit or not. We only say it's unique if the digit doesn't exist in the rest. We do this for every digit. - In the other, we loop through the numbers 0-9 and just call `has_digit` on each one. If it returns true then we know the entire number contains that digit and we can one to our unique count.

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.

One key idea in environment diagrams is the **frame**. A frame helps us keep track of what variables have been defined in the current execution environment, and what values they hold. The frame we start off with when executing a program from scratch is what we call the **Global frame**. Later, we'll get into how new frames are created and how they may depend on their parent frame.

Here's a short program and its corresponding diagram (only visible on the online version of this worksheet):

See the web version of this resource for the environment diagram.

Remember that programs are mainly just 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.

Q8: Assignment Diagram

Use these rules to draw an environment diagram for the assignment statements below:

See the web version of this resource for the environment diagram.

We first assign `x` to the result of evaluating `11 % 4`. We then bind `y` to the current value of `x` (which we can figure out by looking it up in our current environment diagram). Finally, we'd like to update `x` to the new value that is the result of the current `x` squared.

Video walkthrough

def Statements

A `def` statement creates ("defines") a function object and binds it 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.

A very important note: Assignments for `def` statements use pointers to functions, which can have different behavior than primitive assignments (such as variables bound to numbers).

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.

Q9: def Diagram

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

See the web version of this resource for the environment diagram.

We first define the two functions `double` and `triple`, each bound to their corresponding name. In the next line, we assign the name `hat` to the function object that `double` refers to. Finally, we assign the name `double` to the function object that `triple` refers to.

Video walkthrough