1 Review

Last time:

- Last week's HW was using Jupyter notebook. Any questions on Jupyter?
- Basic Python types and operations.
- Conditionals

2 Conditionals Example

Fizz Buzz! You are given a number `num`. Your program should:

1. Print 'fizz' if divisible by 3
2. Print 'buzz' if divisible by 3
3. Print 'fizzbuzz' if divisible by 3 and 5
4. Print the number otherwise

```
num = 15 # some number
d3 = (num%3)==0
d5 = (num%5)==0
if d3 and d5:
    print('fizzbuzz')
elif d3:
    print('fizz')
elif d5:
    print('buzz')
else:
    print(num)
```

3 Loops

- We often want to run a piece of code many times
- While and for loops

While loop: runs WHILE a condition true

```
import random as rand
num = 0
while num!=5:
    print('The number is ' + str(num))
    num = rand.randint(1,10) # random integer between 1-10

print('The number is 5!')
```

For loop: iterates through a type of object called ITERABLE. There are two types of iterable objects that we're concerned with,

1. range(a, b) = $\{x : a \leq x < b\}$
2. range(b) = $\{0 : a \leq x < b\}$
3. lists

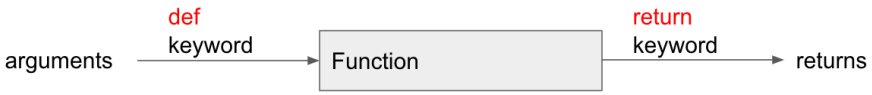
```
# prints 1, 2, ..., 10
for i in range(1,11):
    print(i)

# prints contents of the list
lst = [3, 1, 4, 1, 5]
for i in lst:
    print(i)

# double the contents of a list
for i in range(len(lst)):
    lst[i] = lst[i]*2
```

4 Functions

- Perform a repetitive task that reacts to some input and has some output
- Functions are black boxes. Someone can hand you a function they wrote and you don't have to understand HOW it works but only WHAT it does.



- The name of a function is always followed by `()`
- Defining a function:

```
def say_hi():
    print('hi!')
```

- Calling a function:

```
>>> say_hi()
>>> hi!
```

- Functions can take arguments: information that you pass to the code inside the function.

```
def greet(name):
    print('Hi {}, how are you today?'.format(name))

>>> greet('James')
>>> Hi James, how are you today?
```

- Function can return arguments:

```
def add(x, y):
    sum = x+y
    return sum

>>> result = add(1, 2)
>>> print(result)
>>> 3
```

- Notice that the order of your arguments matters!
- If no return statement if provided, Python returns None by default
- Default arguments: you can specify that an argument should have a default value if not provided by the user.

```
def get_harmonic(freq, harmonic=2):
    return freq*harmonic

>>> get_harmonic(100) #200
>>> get_harmonic(100, 3) #300
>>> get_harmonic(100, harmonic=3) #300, more explicit
```

- Unpacking an iterable: taking an iterable object (e.g. list) and unpacking it's contents.

```
>>> lst = [1, 2]
>>> add(lst[0], lst[1])
>>> 3
>>> add(*lst)
>>> 3
>>> print(*lst)
>>> 1 2
```

- Packing an iterable: what if you want to write a function that takes an unspecified number of arguments?

```
def add(*args):
    sum = 0
    for n in args:
        sum = sum + args
    return sum
```

```
>>> add(1, 2, 3)
>>> 6
>>> add(*lst)
>>> 3
```

Demo: finding all primes up to 1000

- We only need to check factors up to $\lfloor n/2 \rfloor$. Proof: if $x \cdot y = n$, then $y = n/x \leq 2$, which we have already checked.

```
def is_prime(n):
    for i in range(2, int(n/2)):
        if n%i==0:
            return False
    return True

primes = []
for n in range(1, 10001):
    if is_prime(n):
        primes.append(n)

print(*primes, sep=',')
```

5 Errors

There are can be many sources of error in your code:

- syntax
- runtime errors

Whenever an error arises **and is not handled by your code**, your program terminates and a stack trace is initiated.

```
1 # returns if n is divisible by x
2 def is_divisible(n, x):
3       return n%x == 0
4
5 print(is_divisible(1,0))
```

```
>>> Traceback (most recent call last):
  File "demo.py", line 1, in <module>
  File "demo.py", line 2, in f
ZeroDivisionError: integer division or modulo by zero
```

Traceback:

- The traceback traces back the sources of the error
- Read the traceback from bottom to top
- Bottom line: What error caused the code to stop and any messages that accompany the error.
- Moving bottom to top, the most recent line of code where the error occurred.
- Read the traceback to effectively debug code!!

Catching exceptions: sometimes, when an error occurs, you want your code to move on.

```
try:
    print(is_divisible(1,0))
except ZeroDivisionError:
    print('There was a zero division error!')

print(is_divisible(3,2))

>>> There was a zero division error!
>>> False
```

- to handle any error, catch `Exceptions`

Raising exceptions:

- Sometimes, you want your code to cause exceptions. sometimes, you want your code to cause exceptions.
- E.g. want to make sure that the arguments that the user passes to your function make sense
- You can cause an error with the `raise` keyword
- Assertion error: assert that a condition is true, otherwise an `AssertionError` is thrown.

```
def dogs(name):
    if type(name) != str:
        raise TypeError("Dog's name is not a string.")

    assert len(name)>0, "Name your dog, you bastard!"

    print("Your dog's name is " + name)
```

6 Next Week

- A miscellany of topics
- List comprehensions, dictionaries, lambda functions, importing, pip