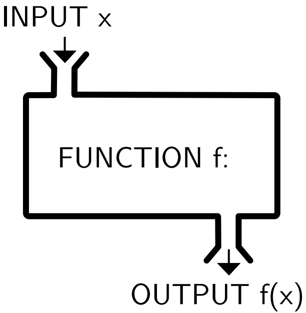
COMP105 Lecture 2

What is a pure function?

Functions



A **function** takes inputs and produces outputs

For example, the function $\texttt{square}(x) = x^2$:

$$\texttt{square}(1) = 1,\ \texttt{square}(2) = 4,\ \texttt{square}(3) = 9,\ \ldots$$

Functions in imperative languages

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

```c
int plus_one(int x)
{
    return x + 1;
}
```

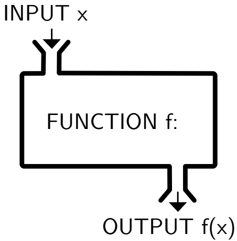
We can implement functions in imperative languages

- But what is known as a "function" can do much more
- We will call these **subroutines**
- Different from **functions** in mathematics

Every function can be implemented as a subroutine,
some subroutines are **not** functions

So what is a pure function?

The **only** thing that matters are the inputs and outputs



We can treat a pure function as a **black box**

- Maybe it is computed by a program
- Maybe a magic wizard answers the question
- We don't care!

But if we use a subroutine we have to care . . .

Side effects

```python
def store_square(x):
    global store
    store = x
    return x * x
```

```c
int polite_plus_one(int x)
{
    printf("Hello there!");
    return x + 1;
}
```

A **side effect** is anything that changes **global state**

- This is anything that can be viewed outside the subroutine
- eg. modifying global variables, printing, network access . . .

Rule:

Pure functions **only** influence the world through return values

When does this matter?

Consider the code: `y = f(1) + f(2)`

Can we rewrite this as: `y = f(2) + f(1)` ?

When does this matter?

Consider the code: `y = f(1) + f(2)`

Can we rewrite this as: `y = f(2) + f(1)` ?

If `f` is a pure function then **yes**

If `f` is a subroutine then **not necessarily!**

- What if `f(1)` means “open file” and `f(2)` means “close file”?
- What if `f` saves its argument to a global variable?
- What if `f` prints is arguments out . . .

Side effects: worked example

```
def f(x):
    print x
    return 2 * x + 1
```

```
>>> y = f(1) + f(2)
1
2
>>> y = f(2) + f(1)
2
1
```

Side effects: the issues

If a sub-routine has side effects then all bets are off

- `f(a) + f(b)` may not be equal to `f(b) + f(a)`
- `f(a) + f(a)` may not be equal to `2 * f(a)`
- We can't necessarily parallelize `f(a) + f(b)`

This is a real issue for

- Code refactoring
- Compiler designers
  - Optimzation
  - Parallelization

If all subroutines are pure functions then these problems go away

Return values

Rule: Pure functions **always** return a value

Subroutines are allowed to have the “void” type

```
void do_something(int x, int y)
{
    // do some stuff
    return;
}
```

Pure functions cannot do this

- The only thing that comes out of a function is the return value
- So what would be the point of a void pure function?

Determinism

Rule: Pure functions must be **deterministic**

- Must give the same answer for the same arguments
- So if `f(1) = 10` then it is always 10

Subroutines can **violate** this, eg., `random.randint(0, 1)`

- Sometimes returns 0, sometimes returns 1
- So not deterministic
- (Actually implemented through side effects)

Determinism

Non-determinism leads to the same problems as before

Does `f(x) + f(x) == 2 * f(x)`?

Determinism

Non-determinism leads to the same problems as before

Does `f(x) + f(x) == 2 * f(x)`?

Not if `f(x) = random.randint(0, x)`!

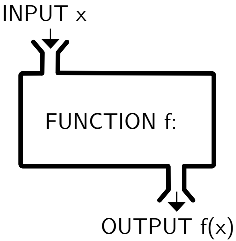
- `f(1) + f(1)`

Outcome	Probability
0	25%
1	50%
2	25%

- But `2 * f(1)`

Outcome	Probability
0	50%
2	50%

Pure functions – summary



Pure functions

- Are a black box
- Have no side effects
- Are deterministic

Every pure function is a subroutine,

some subroutines are not pure functions

Exercise

```python
import string

f = open("input", "r")
l = f.readline()
u = string.upper(l)
l = string.lower(l)
print u, l
```

Which of these subroutines are pure functions?