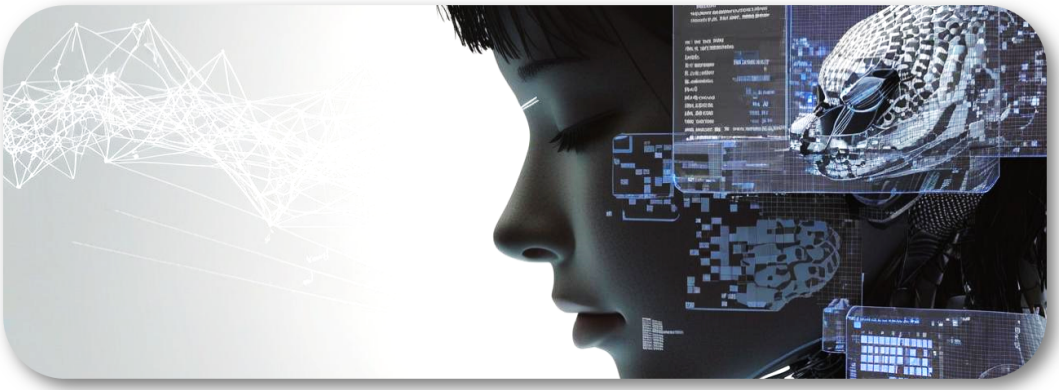

COMP 1023 Introduction to Python Programming

Tutorial 9: More on Functions & Recursion

COMP 1023 Teaching Team

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Quick Refresher - Functions

A function is a collection of statements grouped together that performs an operation.

```
def <function_name>(<parameter_list>):
    <statement_1>
    <statement_2>
    ...
    <statement_N>
```

where <function_name> is the name of the function, <parameter_list> is a list of parameters (i.e., a number of variables), and <statement_1>, ..., <statement_N> denote the program statements that need to be executed when the function is called/invoked.

Example

```
def add_and_multiply(x, y):
    print(x + y)
    return x + y, x * y
```

Variable-Length Parameters

- `*args`: Accept variable number of positional arguments
  - Collected into a tuple inside the function
- `**kwargs`: Accept variable number of keyword arguments
  - Collected into a dictionary inside the function
- Note: `args` and `kwargs` are just names by convention

Example

```
def example(*args, **kwargs):
    print(args)   # (1, 2, 3)
    print(kwargs) # {'name': 'Alice', 'age': 30}

example(1, 2, 3, name="Alice", age=30)
```

Question

What about `def example(**kwargs, *args):`?

Lambda Functions

- Lambda functions are anonymous expressions, meaning they have no names unless explicitly assigned to a variable.
- A lambda function can take in any non-negative number of arguments, but can only have one expression in its body.
- To type hint a lambda expression, we can import the `Callable` type from the `collections.abc` module.

Example

```
from collections.abc import Callable
add: Callable[[int, int], int] = lambda x, y: x + y
print(add(3, 5)) # 8
```

Using Lambda Functions as Arguments

- Lambda functions are useful when you want to pass them as arguments.

Example

```
from collections.abc import Callable

def apply(
    lst: list[int], func: Callable[[int], int],
    filter_func: Callable[[int], bool]
) -> list[int]:
    return [func(ele) for ele in lst if filter_func(ele)]

def main() -> None:
    my_list = list(range(20))
    print(apply(my_list, lambda x: x * 2, lambda x: x % 2 == 0))
```

Output from `main()`: `[0, 4, 8, 12, 16, 20, 24, 28, 32, 36]`

iPRS Question 1

What is the output of the following program?

```
# Type hinting dropped due to laziness
def apply(lst, func, filter_func):
    return [func(ele) for ele in lst if filter_func(ele)]

if __name__ == "__main__":
    my_list = list(range(10))
    print(apply(my_list, lambda x: x ** 2, lambda x, n=3: x % n == 0))
```

A. `[]` (Empty List)

B. `[0, 9, 36, 81]`

C. Syntax error

D. Runtime error

iPRS Question 1 - Solution

Explanation

Answer: B. `[0, 9, 36, 81]`

This program filters the elements if they are divisible by 3 (`x % 3 == 0`). Then, we square the remaining integers.

Functional Programming - `filter()`, `map()` and `reduce()`

- Our `apply` example actually showcases filtering and mapping.
- `filter()`, `map()` and `reduce()` are common functions used to process elements in collections, especially in the functional programming paradigm.

```
def apply(lst, func, filter_func):
    return [func(ele) for ele in lst if filter_func(ele)]

def apply2(lst, func, filter_func):
    return list(map(func, filter(filter_func, lst))
```

Functional Programming - `filter()`, `map()` and `reduce()` (Cont.)

- The `reduce()` function applies a function to an iterable, to *reduce* it to a single value.

```
from functools import reduce

def my_reduce(lst: list[int]) -> int:
    acc = lst[0]
    for e in lst[1:]:
        acc *= e
    return acc

if __name__ == "__main__":
    my_list = [4, 6, 8]
    print(my_reduce(my_list))
    print(reduce(lambda x, y: x * y, my_list))
    # Both produce 192
```

iPRS Question 2

What is the output of the following program?

```
from functools import reduce

def func(lst, mapper, predicate, accumulator):
    return reduce(accumulator, map(mapper, filter(predicate, lst)))

if __name__ == "__main__":
    print(func(
            list(range(11)), lambda e: e + 1 * 2,
            lambda e: e % 2 == 1, lambda a, b: a + b
    ))
```

A. 35

B. 60

C. Syntax error

D. Runtime error

iPRS Question 2 - Solution

Explanation

Answer: A. 35

This program filters the odd elements of `lst`. Then, each element is mapped by adding 1 to the result. Then, the `reduce()` function calculates their sum.

	`lst`
Original	`[0, 1, 2, ..., 9, 10]`
After `filter()`	`[1, 3, 5, 7, 9]`
After `map()`	`[3, 5, 7, 9, 11]`
After `reduce()`	$3 + 5 + 7 + 9 + 11 = 35$

You may notice that the words `mapper`, `predicate`, `accumulator` were used as function parameter names. In fact, these are common names used for mapping, filtering and reducing operations respectively.

Practice Questions - Functional Programming I

```python
from functools import reduce

def func(lst, mapper, predicate, accumulator):
    return reduce(accumulator, map(mapper, filter(predicate, lst)))

def q1():
    lst = list(range(100))
    return func(lst, lambda e: e // 5, lambda e: not e % 10, lambda a, b: a + b)
def q2():
    lst = list(range(100))
    return func(lst, lambda e: e // 2, lambda e: e % 5 == 4, lambda a, b: a + b)
def q3():
    lst = list(map(lambda e: e / 3 + 1, range(1000)))
    return func(lst, lambda e: e % 2, lambda e: e * 2 < 11, lambda a, b: a * b)

if __name__ == "__main__":
    print(q1()) # Part (a)
    print(q2()) # Part (b)
    print(q3()) # Part (c)
```

Practice Questions - Functional Programming I (Answers)

(a) 90

(b) 510

(c) 0.0 (This is a `float`)

Tip

When you see a complicated expression, first check if there are sneaky things in the question. For part (c), you can see the accumulator is multiplication. Since we know 3 is in `lst` and it passed the filter, it will be mapped to 0.0. Since anything times 0 is 0, you can immediately conclude the answer is 0.0!

Practice Questions - Functional Programming II

```python
from functools import reduce

def func(lst, mapper, predicate, accumulator):
    return reduce(accumulator, filter(predicate, map(mapper, lst)))

def q1():
    lst = list(range(10))
    return func(lst, lambda e: e + 1, lambda e: e % 2 == 0, lambda a, b: a + b)
def q2():
    lst = list(range(15))
    return func(lst, lambda e: e / 2, lambda e: e * 3 < 15, lambda a, b: a + b)
def q3():
    lst = list(map(lambda e: e / 3 + 1, range(1000)))
    return func(lst, lambda e: e % 2, lambda _: False, lambda a, b: a + b)

if __name__ == "__main__":
    print(q1()) # Part (a)
    print(q2()) # Part (b)
    print(q3()) # Part (c)
```

Practice Questions - Functional Programming II (Answers)

(a) 30

(b) 22.5

(c) Runtime error, as `reduce()` cannot be applied to an empty iterable.

How do I avoid runtime error with an empty iterable?

If you want to avoid running into runtime error from the `reduce()` function, you can pass in a stating value to the `initial` parameter, e.g., `reduce(accumulator, lst, initial=0)`.

Recursion

- Forming elegant solutions for problems that are difficult to solve using simple loops.
- The process of defining a solution to a problem in terms of a simpler version of itself.
- Recursive functions are functions that call themselves.
- When writing a recursive function, we should define two parts: the base case(s) and the recursive case(s).

Example

```python
def factorial(n: int) -> int:
    """Calculates the factorial of n (n!) recursively."""
    if n == 0 or n == 1:        # Base cases
        return 1
    return n * factorial(n - 1) # Recursive case
```

Recursion Design Patterns

- Pattern 1: Recursive Expression Pattern
  - Define a recursive expression to break problem into smaller parts
  - Return result directly from recursive calls
- Pattern 2: Parameters Passing Pattern
  - Pass a result variable through function parameters
  - Update result during each recursive call
  - Useful for search algorithms and generating combinations

Pattern 1: Recursive Expression Template

- Structure: Define how to solve problem using smaller subproblems
- Base case: Termination condition that returns known value
- Recursive case: Function calls itself with modified arguments

```
def func(args):
    if base_case:           # Termination condition
        return base_value   # Known result
    return recursive_expression  # Call self with modified args

# Example: Factorial uses this pattern
def factorial(n):
    if n == 0 or n == 1:          # Base cases
        return 1                  # Known value
    return n * factorial(n - 1)   # Recursive expression
```

Pattern 2: Parameters Passing Template

- Structure: Pass result variable as function parameter
- Base case: Check condition and output or return
- Recursive case: Modify result and pass to next call

```
def f(args, result):
    if base_case:
        if output_condition:
            print(result)   # Output final result
        return              # End recursion
    f(modified_args, modified_result)  # Pass updated result

def factorial_v2(n, current=0, result=1):
    if current == n:
        print(result)
        return
    factorial_v2(n, current + 1, result * (current + 1))

factorial_v2(5)  # Output: 120
```

Parameter Passing: List Safety

- Problem: Sharing list references between recursive calls
- Risk: Modifying one call affects other calls, causing incorrect results
- Solution: Create new list for each recursive call

```
# GOOD: Create new lists (cost more memory, but safer)
def good_example(lst, pos, result):
    if pos == -1:
        return print(result)  # Same as separate return (print() returns None)
    good_example(lst, pos - 1, result.copy())        # Safe copy
    good_example(lst, pos - 1, [lst[pos]] + result)  # New list

# BAD: Sharing list reference (only use when need less memory occupation)
def bad_example(lst, pos, result):
    if pos == -1:
        return print(result); # Same as separate return (print() returns None)
    bad_example(lst, pos - 1, result)
    result = [lst[pos]] + result
    bad_example(lst, pos - 1, result) # `result` here is different to the previous call!
```

Recursion: Trade-offs and Risks

- Advantages:
  - Elegant solutions for recursive structures (trees, permutations)
  - More concise and readable code for certain problems
- Disadvantages:
  - Higher memory consumption due to call stack
  - Generally slower than iterative solutions
- Risks:
  - Infinite recursion: Missing or unreachable base case
  - Stack overflow: Recursion depth exceeds maximum limit

iPRS Question 3

What is the output of `func("COMP1023")`? Are there potential errors to the function?

```
fil_func = lambda c: c < 'A'
def func(s: str) -> str:
    if len(s) == 1:
        return s if fil_func(s) else ""
    return func(s[1:]) + (s[0] if fil_func(s[0]) else "")
```

A. `3201PMOC`; Yes
B. `3201PMOC`; No
C. `PMOC`; Yes
D. `PMOC`; No
E. `3201`; Yes
F. `3201`; No

iPRS Question 3 - Solution

Explanation

Answer: E. 3201; Yes

This program reverses a string, but filters out characters that are not digits (or more accurately, characters who are before `'A'` in lexicographical order). However, there is a potential runtime error in this program:

- There is no base case accounting for when `s` is an empty string, which would lead to `RecursionError`.

Practice Question - Recursion I

A strictly-increasing integer means every digit in the number is greater than its preceding digit. Write a recursive program `find_int()` that creates all strictly increasing integers with digits 1 to 9 of length `n`. You can assume `n` is always positive.

- `find_int(2)` returns
  `[12, 13, ..., 19, 23, 24, ..., 29, 34, 35, ... 39, 45, 46, ... 49, 56, 57, ..., 59, 67, 68, 69, 78, 79, 89]`.
- `find_int(8)` returns
  `[12345678, 12345679, 12345689, 12345789, 12346789, 12356789, 12456789, 13456789, 23456789]`.
- `find_int(10)` returns an empty list.

Practice Question - Recursion I (Answer)

```python
def find_int(n: int, base: int = 0) -> list[int]:
    if n == 0:
        return [base]
    ret = []
    for i in range(base % 10 + 1, 10):
        ret.extend(find_int(n - 1, base * 10 + i))
    return ret
```

Explanation

We build the numbers digit-by-digit. We "append" a valid digit to the existing number `base` and call the function recursively (after decreasing `n` by 1). If `n` is 0, that means we have reached enough number of digits, so we return a list with `base` as the only element.

Practice Question - Recursion II

Write a recursive program that checks if a substring `sub` appears in a string `target` at least `n` times. You can assume `target` and `sub` are non-empty, and `n` is non-negative. For example:

- `check_substr("abcabcaaabc", "abc", 2)` returns `True`
- `check_substr("abcabcaaabc", "abc", 3)` returns `True`
- `check_substr("abcabcaaabc", "abc", 4)` returns `False`
- `check_substr("catcowcat", "cat", 2)` returns `True`
- `check_substr("catcowcat", "cow", 2)` returns `False`

```
def check_substr(target: str, sub: str, n: int) -> bool:
    pass
    # Your code here
```

Practice Question - Recursion II (Answer)

```python
def check_substr(target: str, sub: str, n: int) -> bool:
    if len(target) < len(sub):
        return n == 0
    if n == 0:
        return True
    if target.startswith(sub):
        return check_substr(target[1:], sub, n - 1)
    return check_substr(target[1:], sub, n)
```

The `str.startswith(str)` method

In the code above, we used the built-in string method `str.startswith`. Simply speaking, it checks if a string starts with the provided substring. In our code, `target.startswith(sub)` works exactly the same as `my_startswith(target, sub)` below.

```python
def my_startswith(target: str, sub: str):
    return target[:len(sub)] == sub
```

Practice Question - Recursion II (Answer, Cont.)

Explanation

We check if the `target` string starts with the substring `sub`, and recursively call itself without the first character of `target`, and decrease `n` by 1 if the substring is found. When `n` reaches 0, that means the substring occurs at least `n` times.

We have 2 base cases in this function:

1. When `len(target) < len(sub)`, `sub` will never be a substring of `target`, so we can check if `n` is 0 or not.
2. When `n` reaches 0, that means our condition is reached, and we can return `True`.

For our recursive case, we first check if our string starts with the substring. If so, we call itself recursively without `target`'s first character and `n` decreased. Otherwise, we simply call itself recursively without `target`'s first character.

Practice Question - Recursion II (Alternate Answer)

A shorter/more elegant solution:

```
def check_substr(target: str, sub: str, n: int) -> bool:
    if len(target) < len(sub) or not n:
        return not n
    return check_substr(target[1:], sub,
        n - 1 if target.startswith(sub) else n
    )
```

Mutual Recursion

- When two or more functions are defined in terms of each other, they are known as mutually recursive functions.

Example

```
def is_even(n: int) -> bool:
    return True if n == 0 else is_odd(n - 1)

def is_odd(n: int) -> bool:
    return False if n == 0 else is_even(n - 1)
```

Mutual Recursion (Cont.)

Mutual recursion is not applied commonly, but they are useful for certain situations:

- Counting number of leaves in a tree (COMP 2012/COMP 3711)
- Implementation of the minimax algorithm (COMP 2211)
- Implementation of a finite-state machine (COMP 3721)
- Recursive descent parsers (COMP 4121)
- and more...

This slide was an excuse for advertisement! ^_^

iPRS Question 4

What is the output of the following program?

```
from functools import reduce
def even(lst: list[int] | None, depth: int = 0)\
    -> list[int] | int:
    if not lst:
        return []
    if depth % 2 == 0:
        filtered = filter(lambda x: x > 0, lst)
        mapped = map(lambda x: x * 2, filtered)
        return odd(list(mapped), depth + 1)
    return odd(lst, depth + 1)
def odd(lst: list[int] | None, depth: int = 0)\
    -> list[int] | int:
    if not lst:
        return 0
    if depth % 2:
        filtered = filter(lambda x: x <= 8, lst)
        return reduce(lambda x, y: x + y, filtered)
    nested = [lst[i:i+2] for i in range(0, len(lst), 2)]
    return even(nested, depth + 1)
```

```
if __name__ == "__main__":
    data = [1, -2, 3, 0, 4, -1, 5]
    print(even(data))
```

A. 14

B. 16

C. 18

D. 22

E. A runtime error will occur

iPRS Question 4 - Solution

Explanation

Answer: B. 16

Since we call `even()` first with a list, we will always execute the second `if`-clause in the functions.
During the first `even()` call, the list is filtered to `[1, 3, 4, 5]`, then mapped to `[2, 6, 8, 10]`.
Then, in the first `odd()` call, the list is filtered again to `[2, 6, 8]`. The `reduce()` function then reduces it to 16.

That's all!

Any questions?