

# Question – How difficult is the triplet project on a scale 1 – 10 ?

- a) 1 (I'm offended by how trivial the project was)
- b) 2 (very easy)
- c) 3 (a quite standard review exercise)
- d) 4 (not too complicated, got some known concepts repeated)
- e) 5 (good exercise to repeat standard programming techniques)
- f) 6 (had to use more advanced techniques in a familiar way)
- g) 7 (quite complicated, but manageable)
- h) 8 (very abstract exercise, using complicated language constructs)
- i) 9 (very complicated – barely manageable spending all my time)
- j) 10 (this is a research project – could be an MSc thesis/PhD project)
- k) 25 (this is wayyy too complicated for a university course)

# Functions as objects

- lambda
- higher-order functions
- map, filter, reduce

# Aliasing functions – both user defined and builtin

## Python shell

```
> def square(x):  
    return x * x  
  
> square  
| <function square at 0x0329A390>  
  
> square(8)  
| 64  
  
> kvadrat = square  
| kvadrat(5)  
| 25  
  
> kvadrat  
| <function square at 0x0329A390>  
  
> len  
| <built-in function len>  
  
> length = len  
> length([1, 2, 3])  
| 3
```

# Functions as values

## square\_or\_double.py

```
def square(x):  
    return x * x  
  
def double(x):  
    return 2 * x  
  
while True:  
    answer = input('square or double ? ')  
    if answer == 'square':  
        f = square  
        break  
    if answer == 'double':  
        f = double  
        break  
  
answer = input('numbers: ')  
L_in = [int(x) for x in answer.split()]  
L_out = [f(x) for x in L_in]  
print(L_out)
```

f will refer to one of the functions  
square and double refer to  
call the function f is referring to  
with argument x

## Python shell

```
| square or double ? square  
| numbers: 3 6 7 9  
| [9, 36, 49, 81]  
  
| square or double ? double  
| numbers: 2 3 4 7 9  
| [4, 6, 8, 14, 18]
```

# Functions as values and namespaces

## say.py

```
def what_says(name):  
    def say(message):  
        print(name, 'says:', message)  
    return say  
  
alice = what_says('Alice')  
peter = what_says('Peter')  
  
alice('Where is Peter?')  
peter('I am here')
```

## Python shell

```
| Alice says: Where is Peter?  
| Peter says: I am here
```

- `what_says` is a function returning a function (`say`)
- Each call to `what_says` with a single string as its argument **creates a new `say` function** with the current `name` argument in its namespace
- In each call to an instance of a `say` function, `name` refers to the string in the namespace when the function was created, and `message` is the string given as an argument in the call

# Question – What list is printed ?

```
def f(x):  
    def g(y):  
        nonlocal x  
        x = x + 1  
        return x + y  
    return g
```

**a** = f(3)

**b** = f(6)

print([**a**(3), **b**(2), **a**(4)])

a) [7, 7, 10]

b) [7, 9, 8]



c) [7, 9, 9]

d) [7, 9, 12]

e) [7, 10, 10]

f) Don't know

# map

- `map(function, list)` applies the function to each element of the sequence `list`
- `map(function, list1, ..., listk)` requires `function` to take `k` arguments, and creates a sequence with the `i`'th element being `function(list1[i], ..., listk[i])`

## Python shell

```
> def square(x):  
    return x * x  
  
> list(map(square, [1,2,3,4,5]))  
| [1, 4, 9, 16, 25]  
  
> def triple_sum(x, y, z):  
    return x + y + z  
  
> list(map(triple_sum, [1,2,3], [4,5,6], [7,8,9]))  
| [12, 15, 18]  
  
> list(map(triple_sum, *zip(*(1,4,7), (2,5,8), (3,6,9))))  
| [12, 15, 18]
```

# sorted

- A list `L` can be sorted using `sorted(L)`
- A user defined order on the elements can be defined by providing a function using the keyword argument `key`, that maps elements to values with some default ordering

## Python shell


```
> def length_square(p):  
    x, y = p  
    return x**2 + y**2 # no sqrt  
  
> L = [(5, 3), (2, 5), (1, 9), (2, 2), (3, 4)]  
> list(map(length_square, L))  
| [34, 29, 82, 8, 25]  
> sorted(L) # default lexicographical order  
| [(1, 9), (2, 2), (2, 5), (3, 4), (5, 3)]  
> sorted(L, key=length_square) # order by length  
| [(2, 2), (3, 4), (2, 5), (5, 3), (1, 9)]
```



# Question – What list does sorted produce ?

```
sorted([2, 3, -1, 5, -4, 0, 8, -6], key=abs)
```

*(Note: In the original image, the numbers 2, 3, 1, 5, 4, 0, 8, 6 are written below the corresponding elements in the list, indicating their absolute values.)*

- a) [-6, -4, -1, 0, 2, 3, 5, 8]
- b) [0, 2, 3, 5, 8, -1, -4, -6]
-  c) [0, -1, 2, 3, -4, 5, -6, 8]
- d) [8, 5, 3, 2, 0, -1, -4, -6]
- e) [0, 1, 2, 3, 4, 5, 6, 8]
- f) Don't know

**Python shell**

```
> abs(7)
| 7
> abs(-42)
| 42
```

# filter

- `filter(function, list)` returns the subsequence of `list` where `function` evaluates to true
- Essentially the same as  
`[x for x in list if function(x)]`

## Python shell

```
> def odd(x):  
    return x % 2 == 1  
  
> filter(odd, range(10))  
| <filter object at 0x03970FD0>  
> list(filter(odd, range(10)))  
| [1, 3, 5, 7, 9]
```

# reduce (in module functools)

- Python's "reduce" function is in other languages often denoted "fold"

$$\text{reduce}(f, [x_1, x_2, x_3, \dots, x_k]) = f(\dots f(f(x_1, x_2), x_3) \dots, x_k)$$

Python shell

```
> from functools import reduce  
  
> def power(x, y):  
    return x ** y  
  
> reduce(power, [2, 2, 2, 2, 2])  
| 65536
```

# lambda (anonymous functions)

- If you need to define a *short* function, that *returns a value*, and the function is only *used once* in your program, then a lambda function might be appropriate:

`lambda arguments: expression`

- Creates a function with no name that takes zero or more arguments, and returns the value of the single expression

Python shell

```
> f = lambda x, y : x + y
> f(2, 3)
| 5
> list(filter(lambda x: x % 2, range(10)))
| [1, 3, 5, 7, 9]
```

# History of lambda in programming languages

- Lambda calculus invented by Alonzo Church in 1930s
- At the heart of functional programming languages
  - LISP (1958)
  - ML (1973) and its derivatives, e.g. OCaml (1996) and F\* (2011)
  - Haskell (1990), based on Miranda (1985)
- C++ got lambdas with C++11 in 2011 (26 years after 1st release)
- Java first got lambdas with Java 8 in 2014 (19 years after 1st release)
- Python has had lambdas since its inception in 1994

# Examples: sorted using lambda

## Python shell

```
> L = [ 'AHA', 'Oasis', 'ABBA', 'Beatles', 'AC/DC', 'B. B. King', 'Bangles', 'Alan Parsons']

> # Sort by length, secondary after input position (default, known as stable)
> sorted(L, key=len)
| ['AHA', 'ABBA', 'Oasis', 'AC/DC', 'Beatles', 'Bangles', 'B. B. King', 'Alan Parsons']

> # Sort by length, secondary alphabetically
> sorted(L, key=lambda s: (len(s), s))
| ['AHA', 'ABBA', 'AC/DC', 'Oasis', 'Bangles', 'Beatles', 'B. B. King', 'Alan Parsons']

> # Sort by most 'a's, if equal by number of 'b's, etc.
> sorted(L, key=lambda s: sorted([a.lower() for a in s if a.isalpha()]))
| ['Alan Parsons', 'ABBA', 'AHA', 'Beatles', 'Bangles', 'AC/DC', 'Oasis', 'B. B. King']

> sorted([a.lower() for a in 'AC/DC' if a.isalpha()])
| ['a', 'c', 'c', 'd']
```

# min and max

- Similarly to `sorted`, the functions `min` and `max` take a keyword argument `key`, to map elements to values with some default ordering

## Python shell

```
> max(['w', 'xyz', 'abcd', 'uv'])
| 'xyz'
> max(['w', 'xyz', 'abcd', 'uv'], key=len)
| 'abcd'
> sorted([210, 13, 1010, 30, 27, 103], key=lambda x: str(x)[::-1])
| [1010, 210, 30, 103, 13, 27]
> min([210, 13, 1010, 30, 27, 103], key=lambda x: str(x)[::-1])
| 1010
```

# defaultdict (from module collections)

- An extension of the built-in dict that automatically initializes undefined items on access by calling a function (factory) to produce a default value to be inserted

`defaultdict` (*function to create default value*, *normal dict arguments*)

## Python shell

```
> scores = {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2]}
> scores['Gladstone'] # access to undefined key in a standard dictionary
| KeyError: 'Gladstone'
> from collections import defaultdict
> scores = defaultdict(lambda : [], {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2]})
> scores
| defaultdict(<function <lambda> at 0x0000026460F0BBE0>, {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2]})
> scores['Gladstone'] # calls lambda without arguments to initialize scores['Gladstone']
| []
> scores
| defaultdict(<function <lambda> at ...>, {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2], 'Gladstone': []})
> scores = defaultdict(list, Mickey=[2, 3, 1], Goofy=[1, 0, 2])
> scores['daisy'].append(7) # calls list() to initialize scores['daisy']
> scores
| defaultdict(<class 'list'>, {'Mickey': [2, 3, 1], 'Goofy': [1, 0, 2], 'daisy': [7]})
```



## polynomial.py

```
def linear_function(a, b):  
    return lambda x: a * x + b  
  
def degree_two_polynomial(a, b, c):  
    def evaluate(x):  
        return a * x**2 + b * x + c  
    return evaluate  
  
def polynomial(coefficients):  
    return lambda x: sum([c * x**p for p, c in enumerate(coefficients)])  
  
def combine(f, g):  
    def evaluate(*args, **kwargs):  
        return f(g(*args, **kwargs))  
    return evaluate  
  
f = linear_function(2, 3)  
for x in [0, 1, 2]:  
    print(f'f({x}) = {f(x)}')  
  
p = degree_two_polynomial(1, 2, 3)  
for x in [0, 1, 2]:  
    print(f'p({x}) = {p(x)}')  
  
print(f'{polynomial([3, 2, 1])(2) = }')  
  
h = combine(abs, lambda x, y: x - y)  
print(f'{h(3, 5) = }')
```

## Python shell

```
| f(0) = 3  
| f(1) = 5  
| f(2) = 7  
| p(0) = 3  
| p(1) = 6  
| p(2) = 11  
| polynomial([3, 2, 1])(2) = 11  
| h(3, 5) = 2
```

# Question – What value is $h(1)$ ?

`linear_combine.py`

```
def combine(f, g):  
    def evaluate(*args, **kwargs):  
        return f(g(*args, **kwargs))  
  
    return evaluate  
  
def linear_function(a, b):  
    return lambda x: a * x + b  
  
f = linear_function(2, 3)  
g = linear_function(4, 5)  
  
h = combine(f, g)  
  
print(h(1))
```

a) 5

b) 9

c) 16

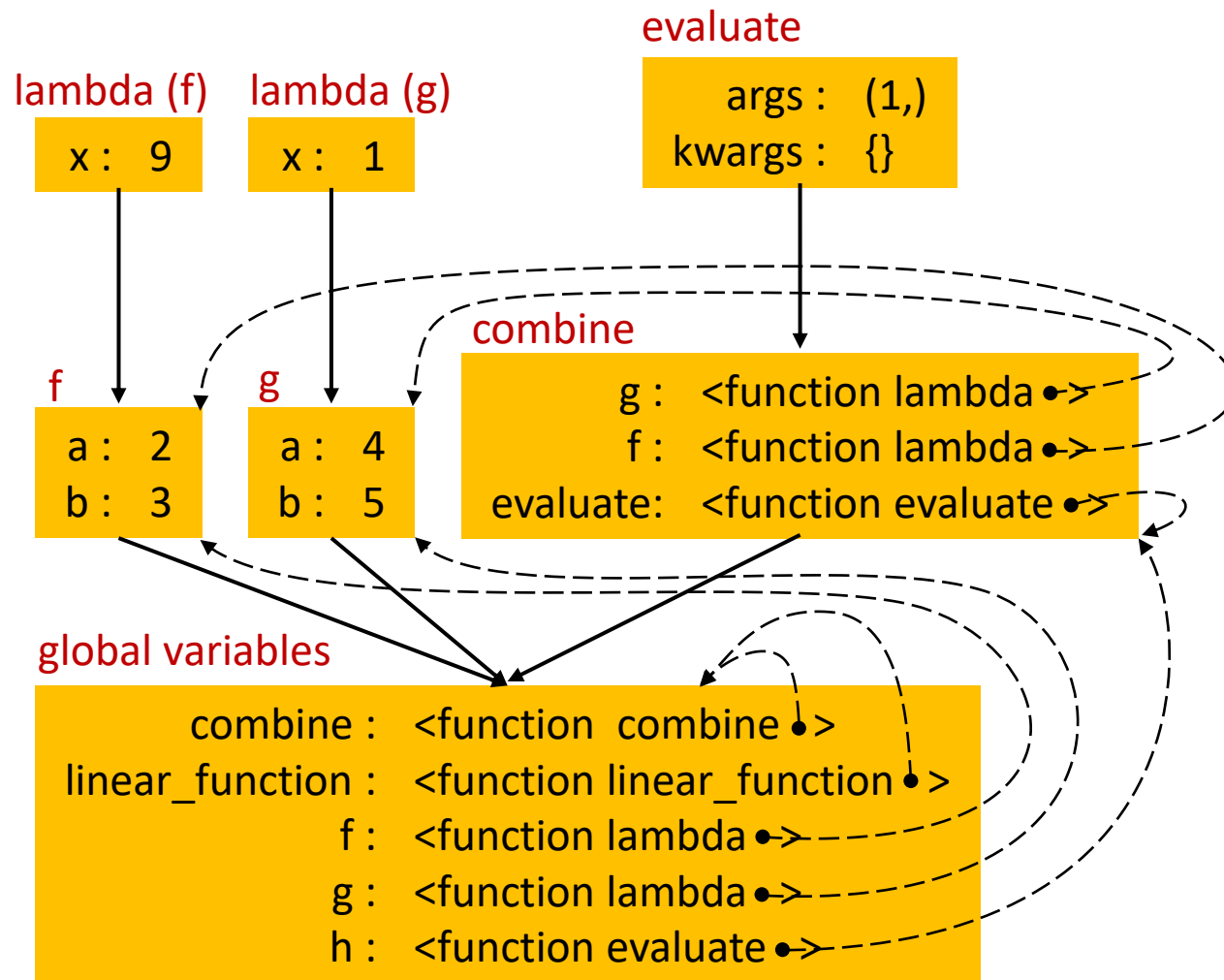


d) 21

e) 25

f) Don't know

# Namespace example



## linear\_combine.py

```
def combine(f, g):  
    def evaluate(*args, **kwargs):  
        return f(g(*args, **kwargs))  
  
    return evaluate  
  
def linear_function(a, b):  
    return lambda x: a * x + b  
  
f = linear_function(2, 3)  
g = linear_function(4, 5)  
  
h = combine(f, g)  
  
print(h(1))
```

# partial (trace of computation)

## partial\_trace.py

```
def partial(fn, *args):
    def new_f(*a):
        print(f'new_f: fn={fn.__name__}, args={args}, a={a}')
        answer = fn(*args, *a)
        print(f'answer={answer}')
        return answer

    return new_f

def f(x, y, z):
    print(f'f({x},{y},{z})')
    return x + 2 * y + 3 * z

g = partial(f, 7)
h = partial(f, 2, 1)
k = partial(g, 1, 2)

print(f'{g(2, 1)=}\n')    # 7 + 2 * 2 + 3 * 1 = 14
print(f'{h(3)=}\n')      # 2 + 2 * 1 + 3 * 3 = 13
print(f'{k()=}\n')       # 7 + 2 * 1 + 3 * 2 = 15
```

## Python shell

```
new_f: fn=f, args=(7,), a=(2, 1)
f(7,2,1)
answer=14
g(2, 1)=14

new_f: fn=f, args=(2, 1), a=(3,)
f(2,1,3)
answer=13
h(3)=13

new_f: fn=new_f, args=(1, 2), a=()
new_f: fn=f, args=(7,), a=(1, 2)
f(7,1,2)
answer=15
answer=15
k()=15
```

## Python shell

```
> def f(x): return x
> g = lambda x: x
> f.__name__
| 'f'
> g.__name__
| '<lambda>'
```

functools.partial