Обектно-ориентирано програмиране

д-р Филип Андонов

16 юни 2022 г.

Съдържание

Обекти

Езиците за програмиране се развиват в посока на повдигане на нивото на абстракция. Целта е програмите да боравят колкото се може повече с термините на решавания проблем и колкото се може по-малко с термините на използвания инструмент. Обектно-ориентираното програмиране (ООП) е средство по пътя към тази цел. В него основна роля играе понятието обект – съставна структура, съдържаща множество променливи (наричани атрибути) и функции(наричани методи) за тяхната обработка, обединени в едно логическо цяло.

Най-често изтъкваните предимства на ООП пред процедурното програмиране са:

- полиморфизъм
- капсулиране
- наследяване

В Python, както и в други езици, дефинирането на нов тип обект се нарича клас. Синтактично това става като започнем с ключовата дума class, след това името на класа и в скоби напишем базовия клас. Всеки клас трябва пряко или непряко да наследява object.

Listing 1: Дефиниране на клас

```
1  class SimpleCalc:
2      def add(self,*a):
3          result = 0
4          for num in a:
5              result += num
6          return result
7
8      def multiply(self,*a):
9          result = 1
10         for num in a:
11             result *= num
12         return result
13
14 calc = SimpleCalc()
15 print(calc.add(1,2,3,100)) #prints 106
16 print(calc.multiply(1,2,3,4)) #prints 24
```

self

Ключовата дума self в Python е референция към текущия обект. Когато в кода извикваме setWeight и getWeight върху ac, ние подаваме ac като първи параметър (това прави self). Интересен в този пример е и факта, че setWeight на практика създава атрибут weight на класа AcmeContainer.

Listing 2: Използване на self

```
1  class AcmeContainer(object):
2      def setWeight(self, weight):
3          self.weight = weight
4
5      def getWeight(self):
6          return self.weight
7
8  ac = AcmeContainer()
9  ac.setWeight(4800)
10 print(ac.getWeight())
```

Конструктор

Обектите трябва да могат да се създават. Тъй като са сложни програмни личности, при създаването им може да е необходимо да се извършат някои действия. Методът със запазено име __init__ се извиква, когато се създава екземпляр от даден клас. Това на практика е конструкторът в Python. Той се използва, за да се извършат инициализациите, които искаме да се изпълнят при създаване на обект. За разлика от други програмни езици, тук дефинирането на член-променливи (атрибути) на класа се извършва не в някаква заглавна част на класа, а именно в конструктора. За целта трябва се използват присвоявания от типа self.attributeName = value

Конструктор

Не бива да изпускаме self, защото именно то указва, че става дума за атрибут. Без него този ред ще създава просто локална променлива. Също така в Python един обект не може да има повече от един конструктор (с различна сигнатура), затова се ползват стойности по подразбиране на параметрите на конструктора. Трябва да имаме предвид, че атрибутите пренаписват методите със същото име. За да се избегне конфликт на имената, може да се използват конвенции за различно именоване на атрибутите и методите. Например, може методите да използват глаголи и имена на атрибути, а атрибутите - съществителни.

Променливи в обектно-ориентирания свят

Данните в класовете и обектите са просто променливи, които са обвързани с пространствата на имената на класовете и обектите, т.е. тези имена са валидни само в контекста на съответните класове и обекти. Съществуват два типа полета в зависимост от това кой ги притежава: променливи на класа и променливи на обекта.

Пространство от имена Class

Променливите на класа са споделени в смисъл, че всеки обект (екземпляр на класа) има достъп до тях. Има само едно копие на променлива на класа, независимо колко обекта има. Когато който и да е обект промени тази променлива, промяната се отразява и на всички останали инстанции.

Променливи на обектите

Променливите на обекта се притежават от всеки индивидуален обект (екземпляр) на класа. Всеки обект има собствено копие на полето, което не е споделено с останалите обекти. Промените в променлива на обекта се отразят само върху този обект, върху който са извършени.

Променливи на класа

Listing 3: Променливи на класа

```
1  class Zombie(object):
2      members = 0
3      def __init__(self,name):
4          self.name = name
5
6      def sayYourName(self):
7          print("Aaargh, ", self.name)
8
9      def summon(self):
10         Zombie.members += 1
```

Променливи на класа

Listing 4: Променливи на класа

```
11 z1 = Zombie("Mariicho")
12 z1.summon()
13 print(Zombie.members)
14 z2 = Zombie("Ivanka")
15 z2.summon()
16 print(Zombie.members)
```

Променливи на класа

В този пример members е променлива на класа, споделена от всички екземпляри и се използва като брояч на създадените инстанции, докато name е атрибут и всеки обект има собствена такава стойност. Когато конструкторът очаква да му се подават параметри, трябва да се внимава със стойностите по подразбиране за съставни типове данни.

Стойности по подразбиране в конструктор

Listing 5: Съставни типове аргументи на конструктора

```
1 class SomeClass(object):
2     def __init__(self,lst=[]):
3         self.lst = lst
4
5     def add2lst(self,elem):
6         self.lst.append(elem)
7
8     def printlst(self):
9         print(self.lst)
```

Стойности по подразбиране в конструктор

Listing 6: Съставни типове аргументи на конструктора

```
2  class Explained(object):
3      lst = []
4      def __init__(self,lst_arg):
5          lst = lst_arg

7      def add2lst(self,elem):
8          self.lst.append(elem)

10     def printlst(self):
11         print(self.lst)
```

Стойности по подразбиране в конструктор

Listing 7: Съставни типове аргументи на конструктора

```
1 class Working(object):
2     def __init__(self):
3         self.lst = []
4
5     def add2lst(self,elem):
6         self.lst.append(elem)
7
8     def printlst(self):
9         print(self.lst)
```

Стойности по подразбиране в конструктор

Конструкторът на SomeClass очаква списък като параметър. Той обаче има стойност по подразбиране, която е празен списък. Когато създадем два обекта от този клас и добавим елемент в атрибута self.lst на единия, то той неочаквано се появява и в другия, сякаш е променлива на класа, а не на обекта. Всъщност точно това се случва. Когато присвояваме стойност на параметър, ние го създаваме, и тъй като той не е обвързан, става променлива на класа. Еквивалентно нещо се случва в класа Explained. Просто lst е променлива на класа.

Стойности по подразбиране в конструктор

Има няколко подхода за решаването на този проблем.

- Можем да направим параметъра на конструктора да няма стойност по подразбиране.
- Друг вариант е да премахнем изобщо списъка като параметър на конструктора, а той да бъде просто атрибут на класа, който може да се попълва със съдържание с нещо като метода add2list на класа Working.
- Трети вариант, макар и малко по-тромав, е конструкторът да приема параметър списък и в кода на конструктора да се проверява дали е подаден наистина списък или стойност None.

Полиморфизъм

Думата произхожда от гръцки и означава „имащ много форми“. Това на практика означава, че дори да не знаем към какъв обект е референция дадена променлива, можем да извършим някакви операции върху него, които ще работят по различен начин в зависимост от типа (класа) на обекта. Полиморфизмът позволява функция да работи по един и същ начин с различни по тип обекти.

Полиморфизъм

Нека имаме система, която изчислява разпределението на масата на някакъв товарен кораб. Ако получаваме данните за контейнерите само от една система на една фирма, за която знаем, че праща данните като редица (номер на контейнер, маса), тогава кодът ще изглежда така:

Listing 8: Първи пример за нуждата от полиморфизъм

```
1  c1 = ('WR24592', 4800)
2  weight = weight+c1[1]
```

Ако горните условия са изпълнени, това е работещ вариант. Но ако работим с още две компании, от които едната предоставя интерфейс към тяхна система, предоставяща масата на контейнерите през мрежата, а другата предоставя данните като низ и шестнайсетично число в речник с ключ weight, нашият код ще се разрастне:

Listing 9: Втори пример за нуждата от полиморфизъм

```
1 def getWeight(object):
2     if isinstance(object, tuple):
3         return object[1]
4     elif isinstance(object, dict):
5         return int(object['weight'])
6     else:
7         return get_weight_over_internet_method(
              object)
```

Полиморфизъм

Вижда се, че авторът на кода трябва да знае по какъв начин да извлича данните за масата на контейнерите и да променя този код всеки път, когато се промени начина на съхранение на тези данни. Ако ползваме обекти, то нас не ни интересува как са реализирани те, стига да знаем как да получим данните за масата. Например, ако знаем, (или нашата система изисква да има публичен метод с това име и тип на връщания резултат), че съществува метод с име getWeight, който връща масата, то ние можем смело да го викаме, без да ни интересува как е представена масата вътре в обекта.

Listing 10: Пример за полиморфизъм

```
1 >>> 'abc'.count('a')
2 1
3 >>> [1, 2, 'a'].count('a')
4 1
```

Полиморфизъм

Полиморфизъм има и във вградените типове данни, както виждаме от примера по-горе. Първо извикваме метод count върху обект от вградения тип стринг, после върху списък. Една и съща операция работи по различен начин върху различните типове обекти, което именно е полиморфизмът.

Listing 11: Пример за полиморфизъм

```
>>> 'abc'.count('a')
1
>>> [1, 2, 'a'].count('a')
1
```

Капсулиране

Капсулирането прилича на полиморфизма, защото и двете са средства за повдигане на абстракцията. То дава възможност да се използва обект, без да се интересуваме как е реализиран. Това е прословутата парадигма за черната кутия, която информатиците толкова много обичат. В случая с обектите идеята е, че ако знаем как да комуникираме с тях, няма нужда да бъдем любопитни и да гледаме как са реализирани (абсолютна лъжа, но я пише във всяка книга, явно помага за продажбите). За да работи по този начин, всеки обект трябва да има собствено състояние (набор от стойности на атрибутите), което да е независимо от каквото и да е извън него.

Капсулиране

Редиците могат да се конкатенират като символните низове (всъщност низовете са редици и Повечето обектно-ориентирани програмни езици предоставят различни модификатори за достъп – public, private, protected. Това, което те казват е – „хей, колега, нямам ти доверие! Подозирам, че като вземеш моя хубав обект ще направиш някакви лоши неща с него, които аз не съм предвиждал, когато съм го съставял. Затова няма да ти разреша да го гледаш. Или ще можеш да го гледаш, но няма да го пипаш.“ В Python на практика такъв механизъм отсъства. Философията на Python е, че се очаква да познаваме един обект преди да го ползваме, например да знаем дали е безопасно да модифицираме атрибут директно отвън.

Наследяване

Понякога при проектирането на нашето приложение ще установяваме, че част от обектите, които искаме да имаме си приличат по много неща и се различават само по някои. В голяма част от случаите това ще се дължи на факта, че те са различни елементи от една обща таксономия. Например, ако една търговска система има продавачи, купувачи и техници, то те ще имат общи атрибути, защото всички те са хора и ние ще искаме да знаем техните имена, контакти и други подобни. Ако имаме система за продажба на МПС, то колите, бусовете и моторите ще имат много общи атрибути, именно защото те всички са МПС-та. В такива случаи е удобно да създадем йерархия от класове, в която всички общи елементи (атрибути и методи) са изнесени в клас – родител, наречен базов, а всички специфични елементи – в съответния наследник.

Listing 12: Наследяване

```
1  class Human(object):
2      def __init__(self,name):
3          self.name = name
4
5      def sayYourName(self):
6          print("Hello, I am ",self.name)
7
8  class Zombie(Human):
9      def sayYourName(self):
10         print("Aaargh , ",self.name)
11
12 h1 = Human("Mariicho")
13 h1.sayYourName()
14 z1 = Zombie("Ivanka")
15 z1.sayYourName()
```

Множествено наследяване

Някои езици допускат един клас да наследява няколко други класа. Това звучи добре, все пак хората също имат повече от един родител. Предимството на този подход е, че лесно могат да се създават хибриди – обекти, които са едновременно две или повече неща. Ако например искаме да създадем клас „клавиатура с тракпад“ и имаме класове клавиатура и тракпад ние можем да създадем новия клас като просто наследим едновременно и двата, от които е съставен. Следният пример създава клас „дете“, което (както и в живота) е наследило не толкова хубавите качества и на двамата си родители.

Listing 13: Множествено наследяване

```
1  class Dad(object):
2      def work(self):
3          print("I am too lazy to work")
4
5  class Mom(object):
6      def acceptBlame(self):
7          print("It is not my fault")
8
9  class Child(Dad,Mom):
10     pass
11
12 Maria = Child()
13 Maria.work()
14 Maria.acceptBlame()
```

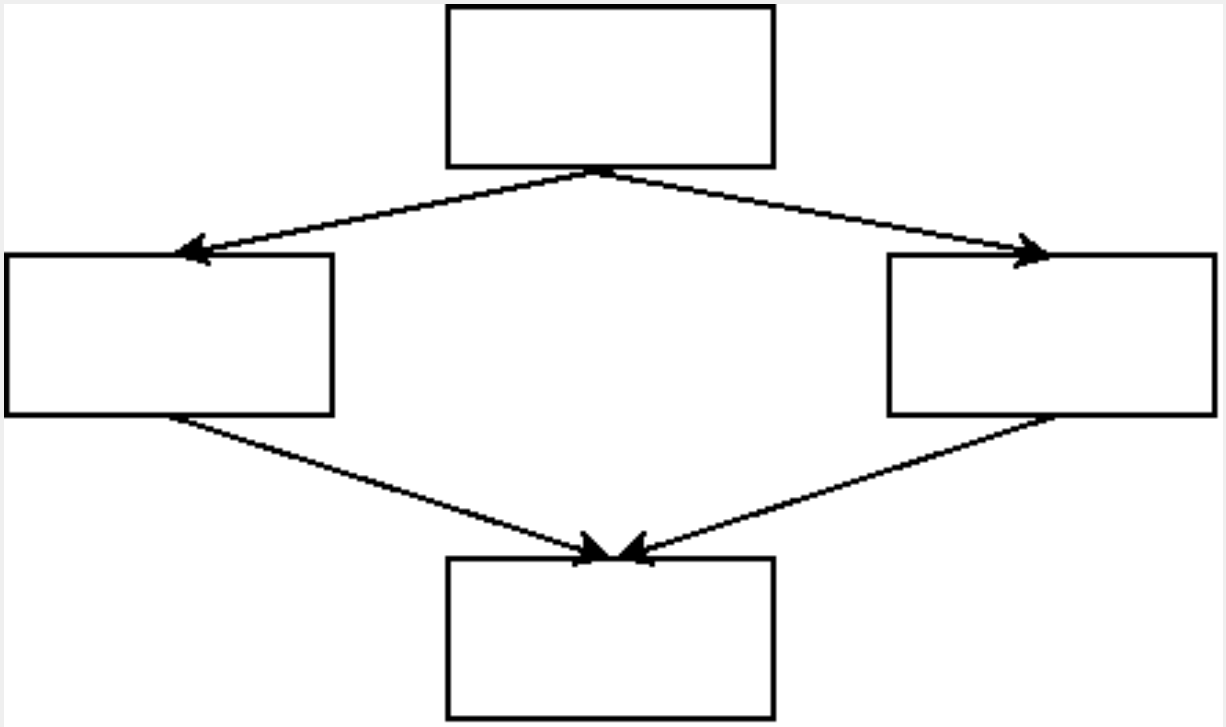
Множествено наследяване

В следващите версии множественото наследяване се реализира от по-сложен алгоритъм, наречен C3 Method Resolution Order. Той работи по следния начин:

- Добавя към списъка всички родители на обекта
- Добавя в края на списъка всички родители на родителите
- Това се повтаря докато има родители
- Ако някой клас се намира в списъка повече от веднъж, премахваме всички появи с изключение на последната.

Множествено наследяване

Проблемът, който прави множественото наследяване не толкова често използваемо (и недопустимо в някои езици), се появява когато базовите класове имат методи или атрибути с едни и същи имена. Така се получава двусмислие и не е ясно какво наследява производния клас. Понякога той се нарича диамантения проблем, поради схемата на наследяване, която го поражда.



Множествено наследяване

В Python няма специални механизми за предпазване от този проблем, както в Go например. Затова общата препоръка е да използваме множествено наследяване само когато наистина се налага и сме сигурни, че няма двусмислие при наследените методи и атрибути.

Интерфейси

В езици като Java съществуват т. нар. интерфейсни класове, които трябва да се наследят и имплементират, за да може да се каже, че обекти от производния клас отговарят на някакъв стандарт. В Python не се определя явно кои методи трябва да реализира даден обект, за да бъде приет като аргумент, просто се предполага, че обектът може да прави това, което е поискано от него. Ако не може, ще се породи изключение и програмата ще завърши неуспешно. Сходна концепция на интерфейсите в Python са протоколите, за които ще стане въпрос по-нататък.

Интроспекция

Интроспекцията е способността да се определя типа и свойствата на обект по време на изпълнение. Това е важно в езици като Python, в които всичко е обект. Ако искаме да видим всички атрибути на даден обект, използваме функцията dir. Тя връща списък от всички методи и свойства на обекта. Можем да използваме функцията type, за да научим типа на даден обект, а за да проверим дали даден обект е от зададен тип използваме isinstance.

Listing 14: Интроспекция

```
>>> dir (list)
['__add__', '__class__', '__contains__', '
    __delattr__', '__delitem__', '__delslice__', '
    __doc__', '__eq__', '__format__', '__ge__', '
    __getattribute__', '__getitem__', '
    __getslice__', '__gt__', '__hash__', '__iadd__
    ', '__imul__', '__init__', '__iter__', '__le__
    ', '__len__', '__lt__', '__mul__', '__ne__', '
    __new__', '__reduce__', '__reduce_ex__', '
    __repr__', '__reversed__', '__rmul__', '
    __setattr__', '__setitem__', '__setslice__', '
    __sizeof__', '__str__', '__subclasshook__', '
    append', 'count', 'extend', 'index', 'insert',
     'pop', 'remove', 'reverse', 'sort']
```

Можем да определим уникалния идентификатор на обект с функцията id. Можем да проверим дали даден атрибут или метод е наличен с hasattr.

Listing 15: hasattr

```
1 print(hasattr(Maria, 'work'))
2 print(hasattr(Maria, 'dance'))
3 >>>true
4 >>>false
```

Интроспекция

За по-сериозна интроспекция можем да използваме модула inspect.

inspect.getmembers(some_object, [предикат])

Това е основната функция на модула. Тя връща списък от всички членове на обекта като списък от двойки (име, стойност). Ако за предикат е дадена стойност на незадължителния параметър ще се върнат само тези членове, за които предикатът е върнал Истина.

ismodule(object)

Булева функция, която връща Истина, ако подаденият параметър е модул.

Интроспекция

isclass(object)
Булева функция, която връща Истина, ако подаденият параметър е клас, независимо дали вграден или дефиниран от потребителя.
ismethod(object)
Булева функция, която връща Истина, ако подаденият параметър е обвързан или необвързан метод в Python.
isfunction(object)
Булева функция, която връща Истина, ако подаденият параметър е функция, включително ламбда изрази.

isgeneratorfunction(object)
Булева функция, която връща Истина, ако подаденият параметър е генераторна функция.
getdoc(object)
Функция, която връща docstring на обект, почистен с cleandoc().
getcomments(object)
Връща низ с всеки ред, който е коментар предшестващ директно кода на обект (клас, функция, метод, модул).
getfile(object)
Връща името на файла, в който е дефиниран обектът, който е подаден като параметър. Ако обектът е вграден, ще се породи изключение от тип TypeError.

getmodule(object)
Опитва се да идентифицира в кой модул е дефиниран даден обект.
getsourcefile(object)
Връща името на изходния файл, в който е дефиниран обекта, подаден като параметър. Ако обектът е вграден, ще се породи изключение от тип TypeError.
getsourcelines(object)
Връща списък от редове код и номера на реда, от който започва обекта. Аргументът може да е модул, клас, метод, функция и др.

Интроспекция

getsource(object)
Връща текста на изходния код на обекта като единичен символен низ. Аргументът може да е модул, клас, метод, функция и др.
cleandoc(doc)
Премахва отместванията от docstring символните низове, които са били отместени за да се подравнят с кода.

Сериализиране

Сериализирането е процес, при който обектната йерархия е конвертирана до поток от байтове, за да бъде съхранен във файл или база данни. Целта е да се съхрани състоянието на обектите и да бъде лесно и бързо възстановено при нужда. Десериализирането е обратният процес. В Python съществува модул, наречен pickle, реализиращ алгоритъм за сериализиране и десериализиране на структурата на python обекти. Pickle има оптимизиран вариант, наречен cPickle, който е реализиран на C и е около 1000 пъти по-бърз. Той обаче не поддържа наследяване на Pickler() и Unpickler() класовете, защото в cPickle те са функции, а не класове. За повечето цели това не е и необходимо, така че използването на оптимизирания модул е оправдано. В Python 3 няма cpickle. Идеята в третата версия е, че няма отделни имена на библиотеките за ускорената версия и за реализираната на Python версия. Вместо това, ускорената версия се опитва да се зареди и само ако това е неуспешно се зарежда чистата Python версия.

Сериализиране

Основните функции в модула са следните:
pickle.dump(obj file [protocol])
Записва сериализирана йерархия във файла, подаден като аргумент. Ако не е подадена стойност за параметъра протокол се използва стойност 0. При отрицателна стойност се използва най-новата версия на протокола.
pickle.load(file)
Прочита сериализирана йерархия от отворен файл. Функцията автоматично определя дали байтовият поток е записан в двоичен режим или не.

Сериализиране

pickle.dumps(obj[, protocol])
Връща сериализираната йерархия от символен низ вместо от файл. Ако не се подаде аргумент за параметъра-протокол, се използва стойност 0. При отрицателни стойности се използва най-новата версия на протокола.
pickle.loads(string)
Прочита обектната йерархия от символен низ.

Сериализиране

За демонстрация, ще създадем сложна структура - речник от различни типове ключове и съответстващи стойности, както и списък, и ще ги сериализираме във файл (редове 7 и 9).

Listing 16: Сериализиране

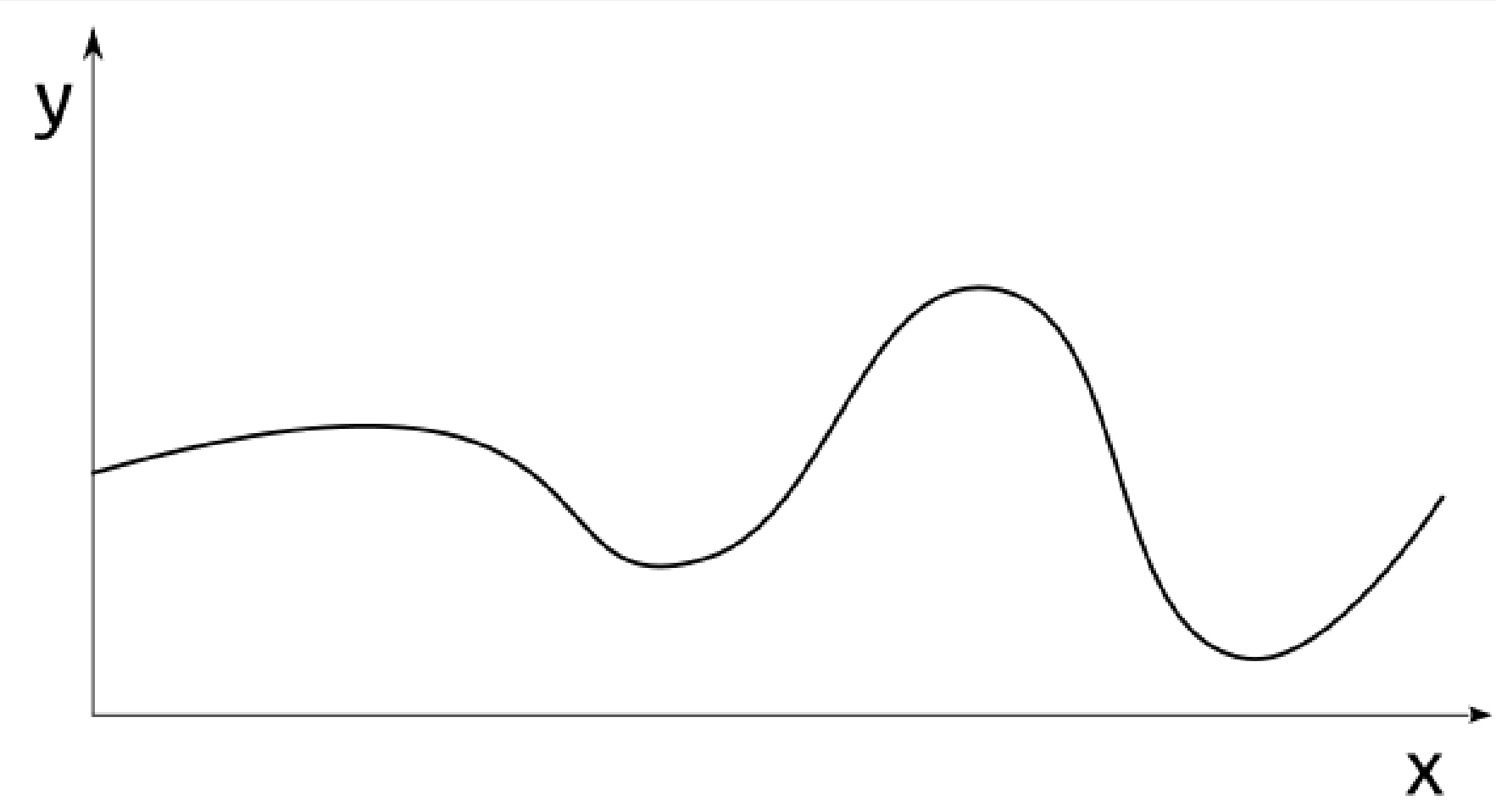
```
1  import pickle
2  data1 = {'a': [1, 2.0, 3, 4+6j], 'b': ('string',
       u'Unicode string'),'c': None}
3  selfref_list = [1, 2, 3]
4  selfref_list.append(selfref_list)
5  output = open('data.pkl', 'wb')
6  #Pickle dictionary using protocol 0.
7  pickle.dump(data1, output)
8  #Pickle the list using the highest protocol
9  pickle.dump(selfref_list, output, -1)
10 output.close()
```

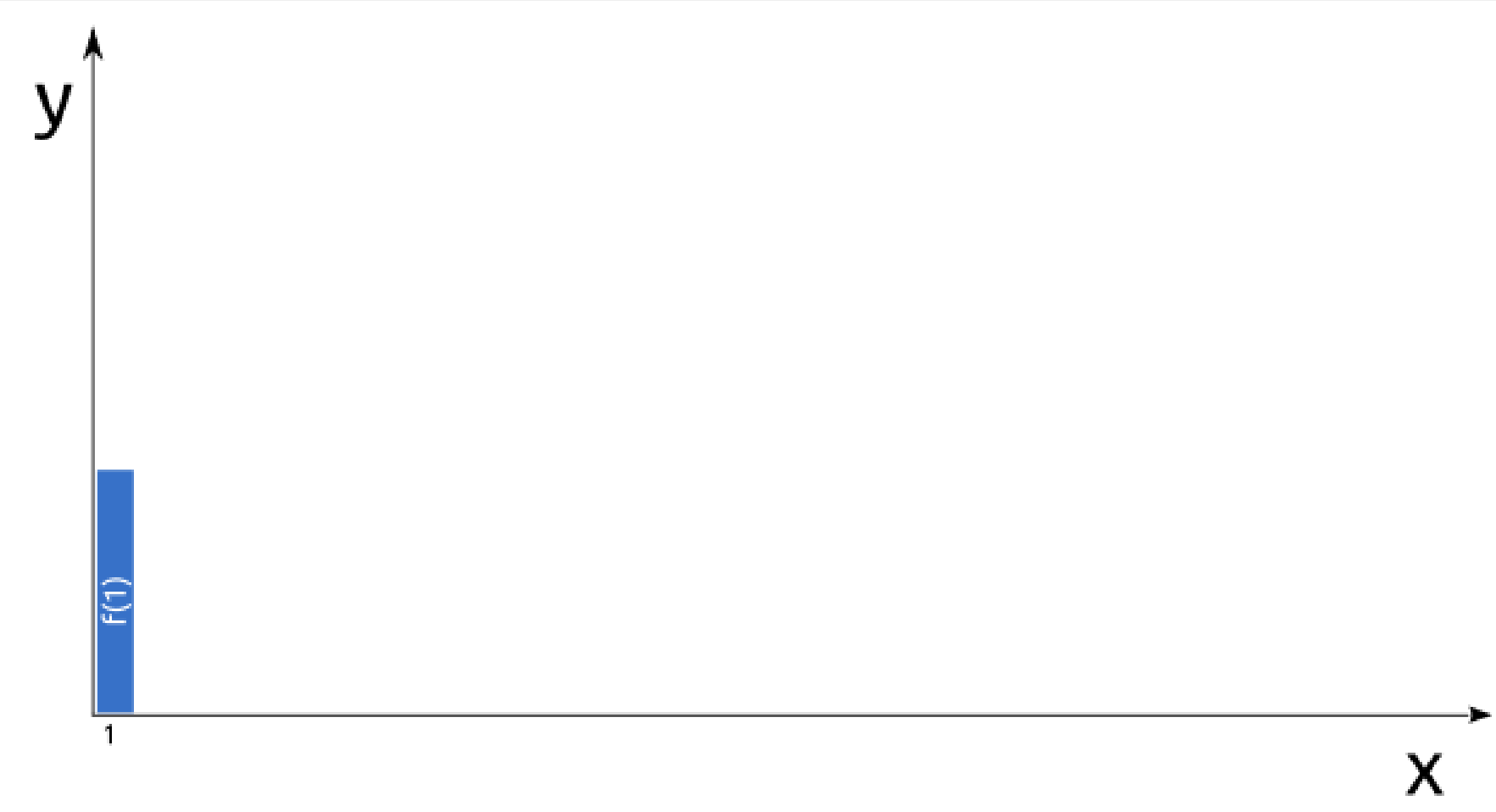
Задачи

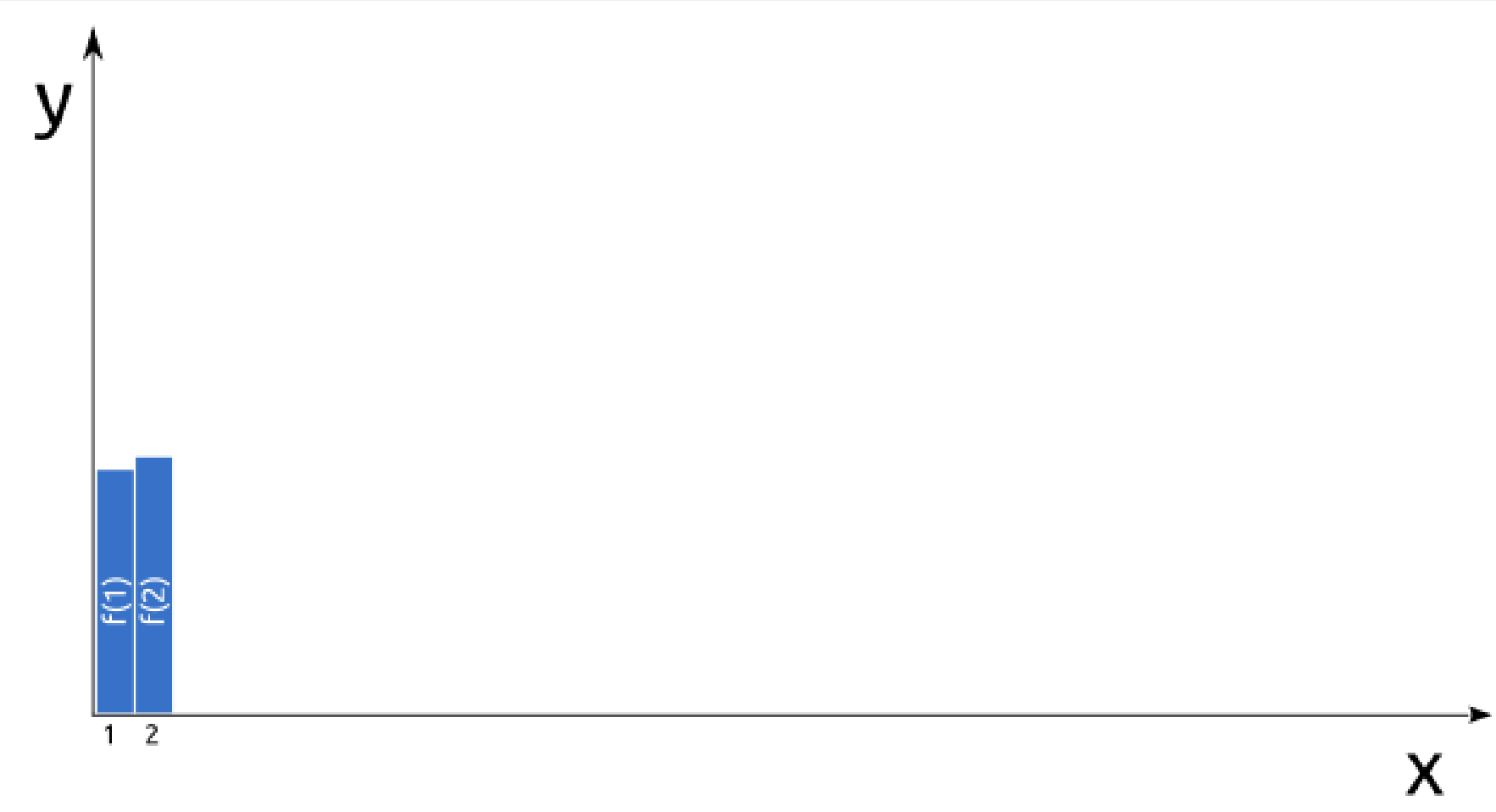
Да се напише програма, реализираща класове SchoolMember, наследени от него класове Teacher и Student, като всеки от тях има атрибути име и възраст. Обектите от класа Teacher трябва също да съхраняват информация за заплатата си и списък от курсовете, които преподават. Обекти от класа Student съхраняват информация за курсовете, които са записали, годината, в която е записан всеки всеки курс и списък с оценки за всеки курс.

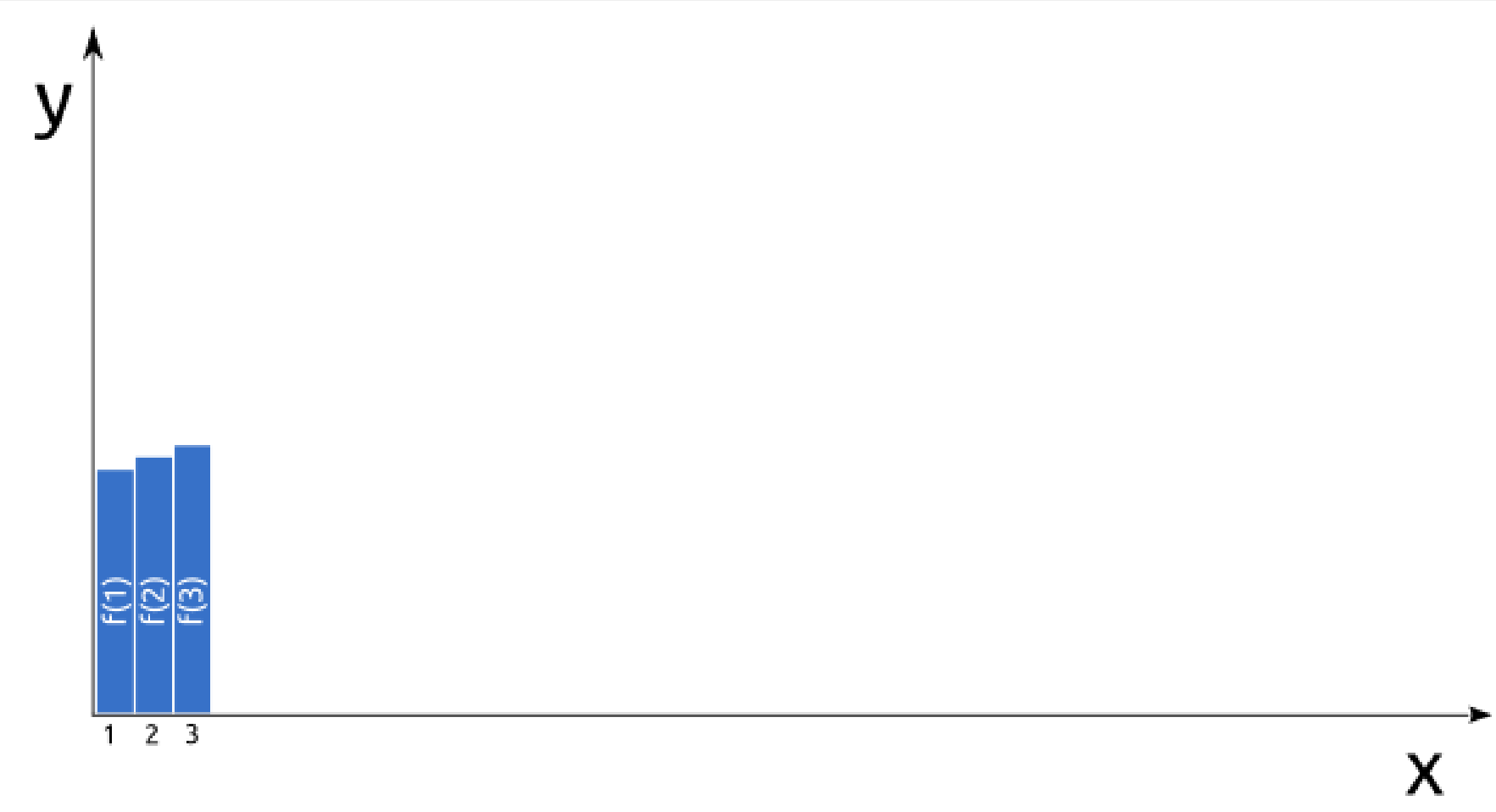
Табулиране на функция

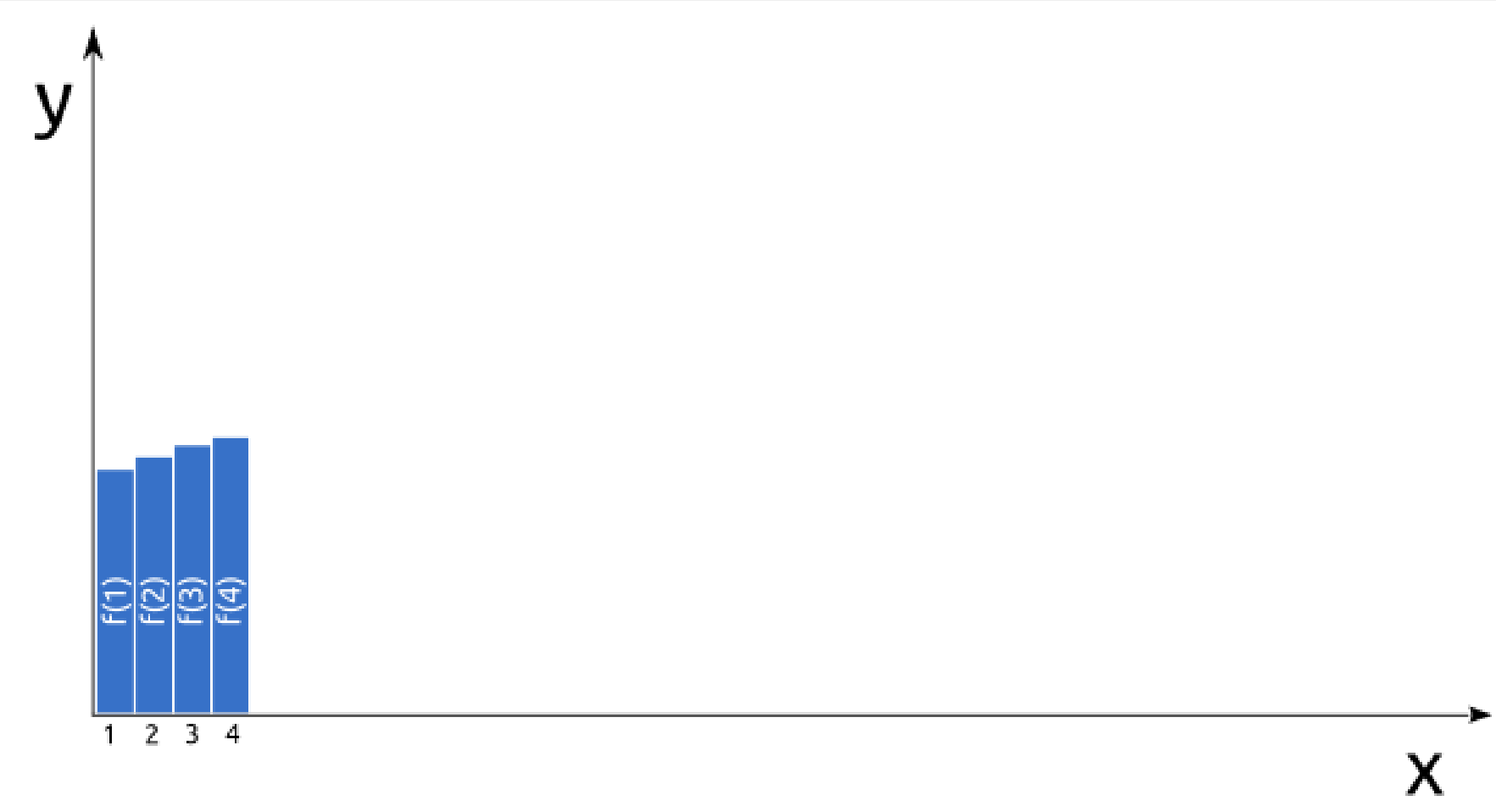
y=f(x)
Стойността y се променя (е функция) на x.
За всяка стойност на x има стойност на y
Когато x е реално число x има безкрайно много стойности и съответни стойности на y.

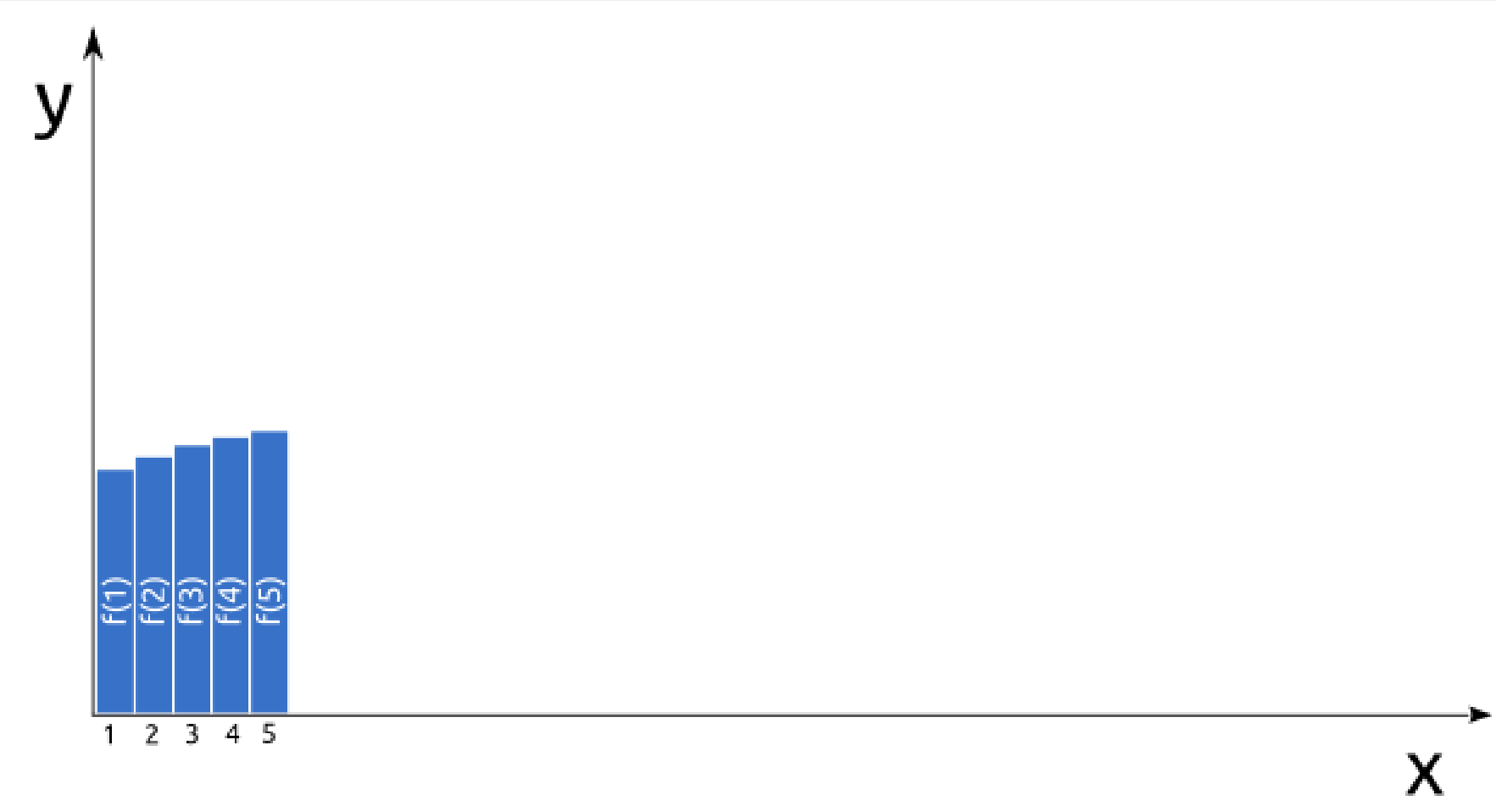

y
x

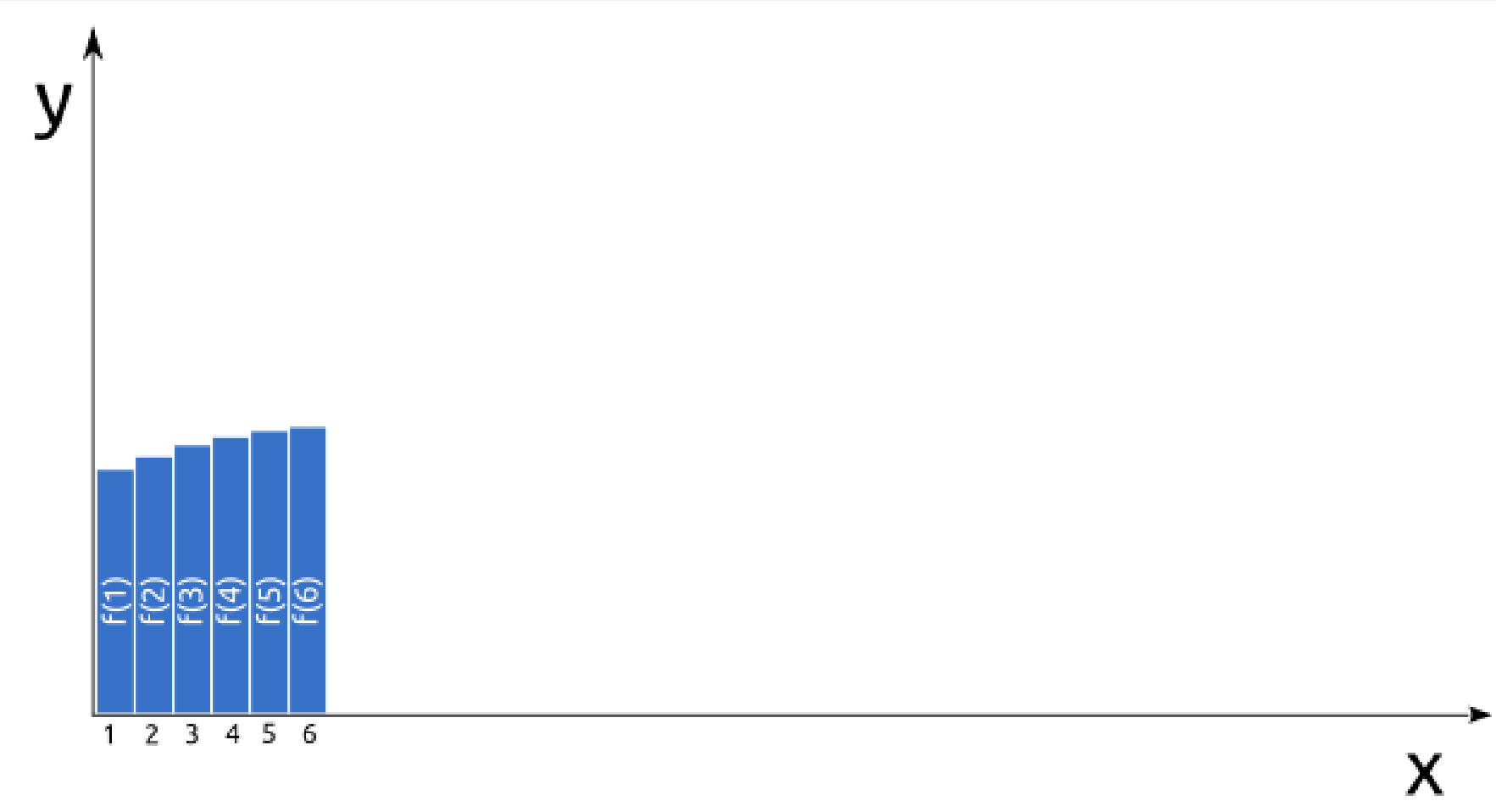

y
F(1)
1
x

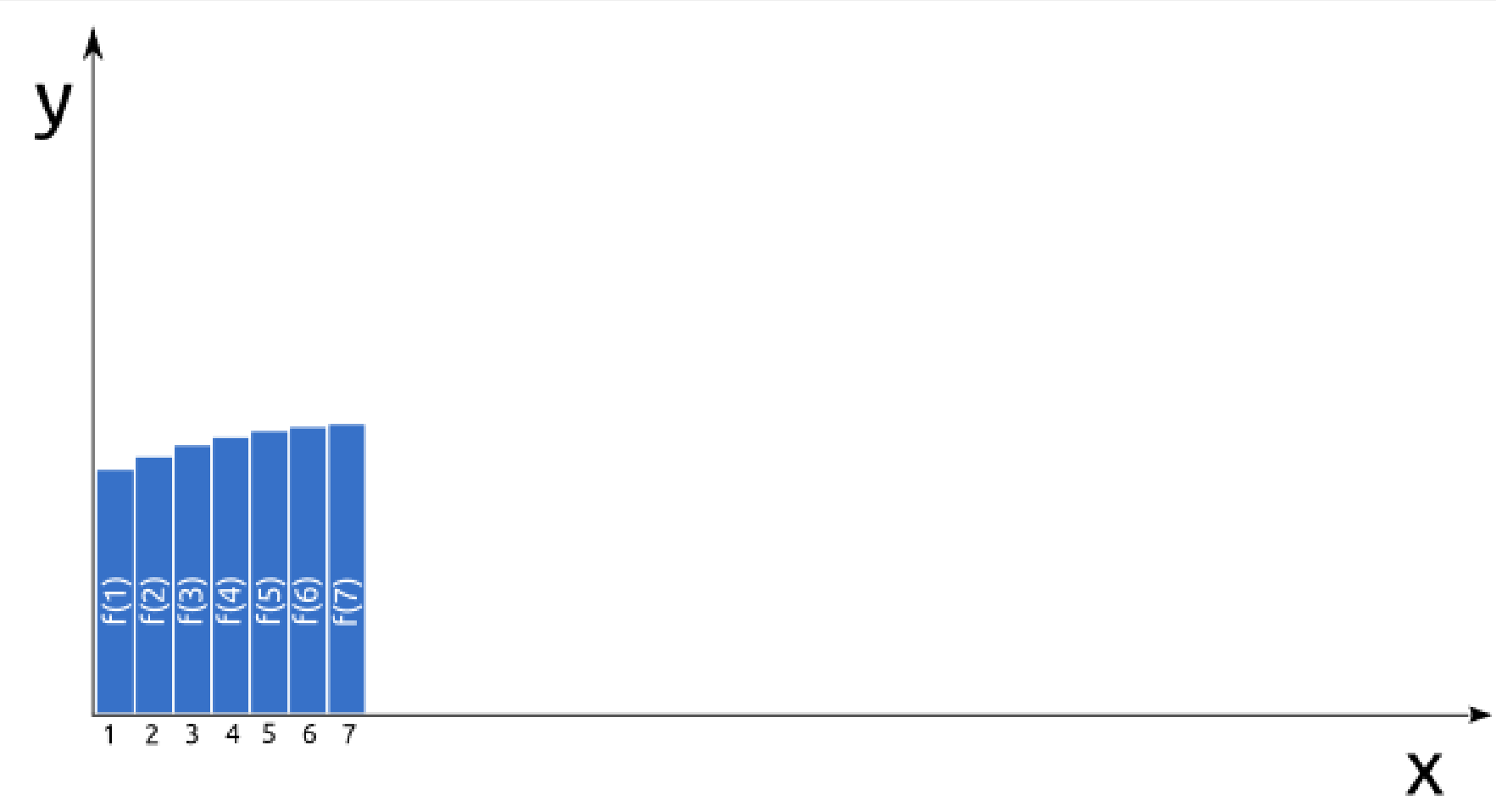

y
F(1)
F(2)
1
2
x

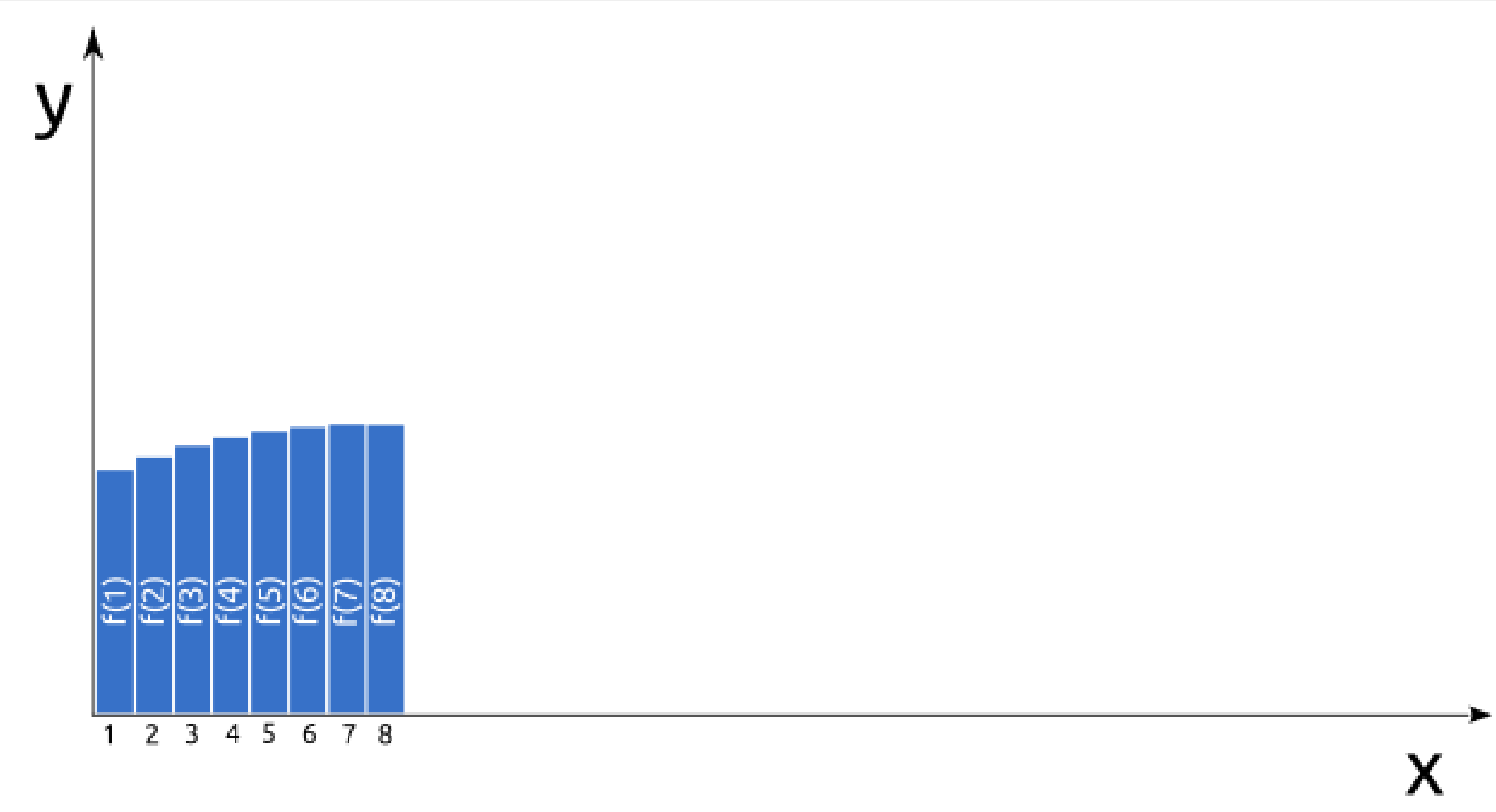

y
f(1)
f(2)
f(3)
1
2
3
x

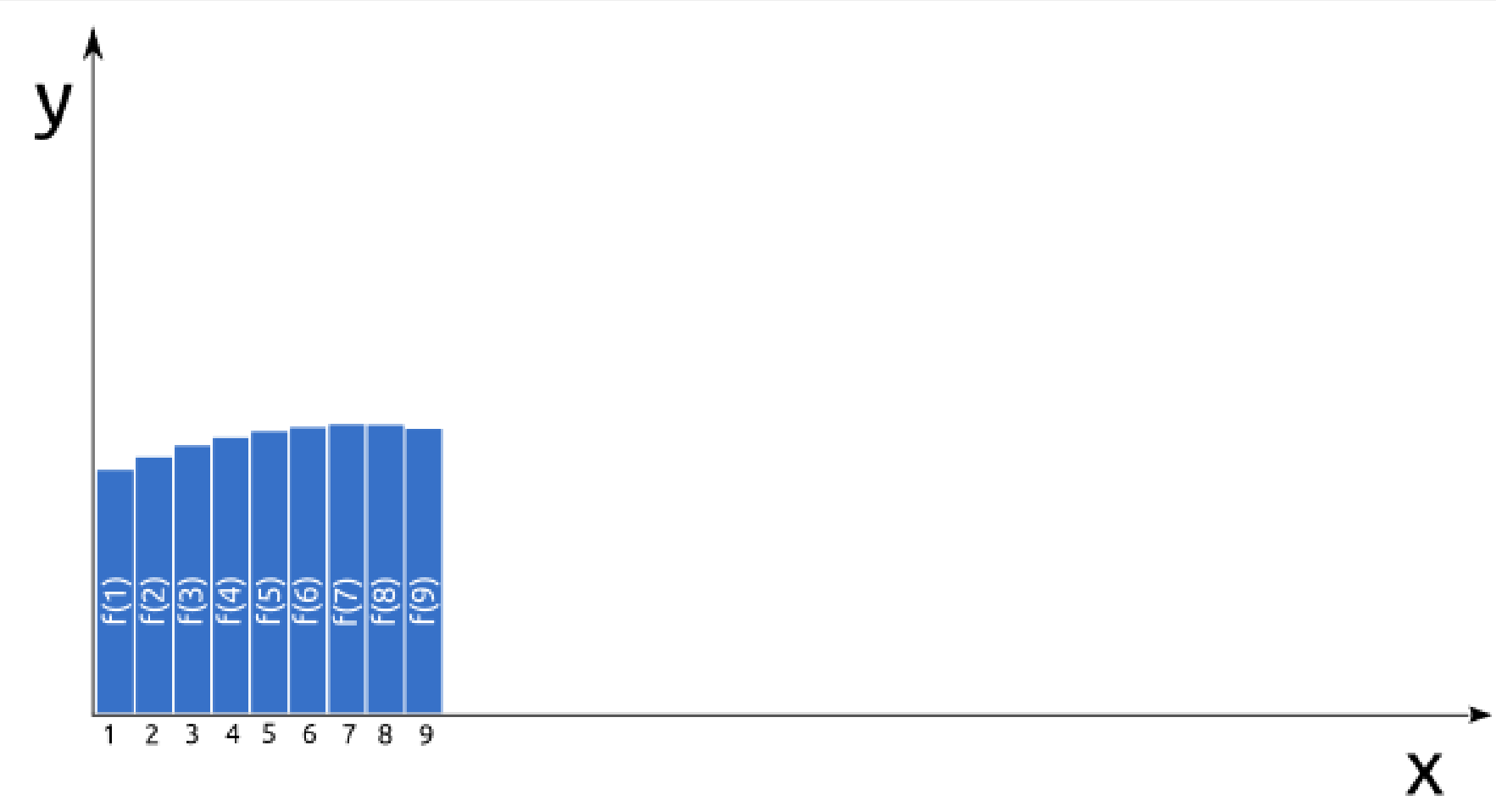

y
f(1)
f(2)
f(3)
f(4)
1
2
3
4
x

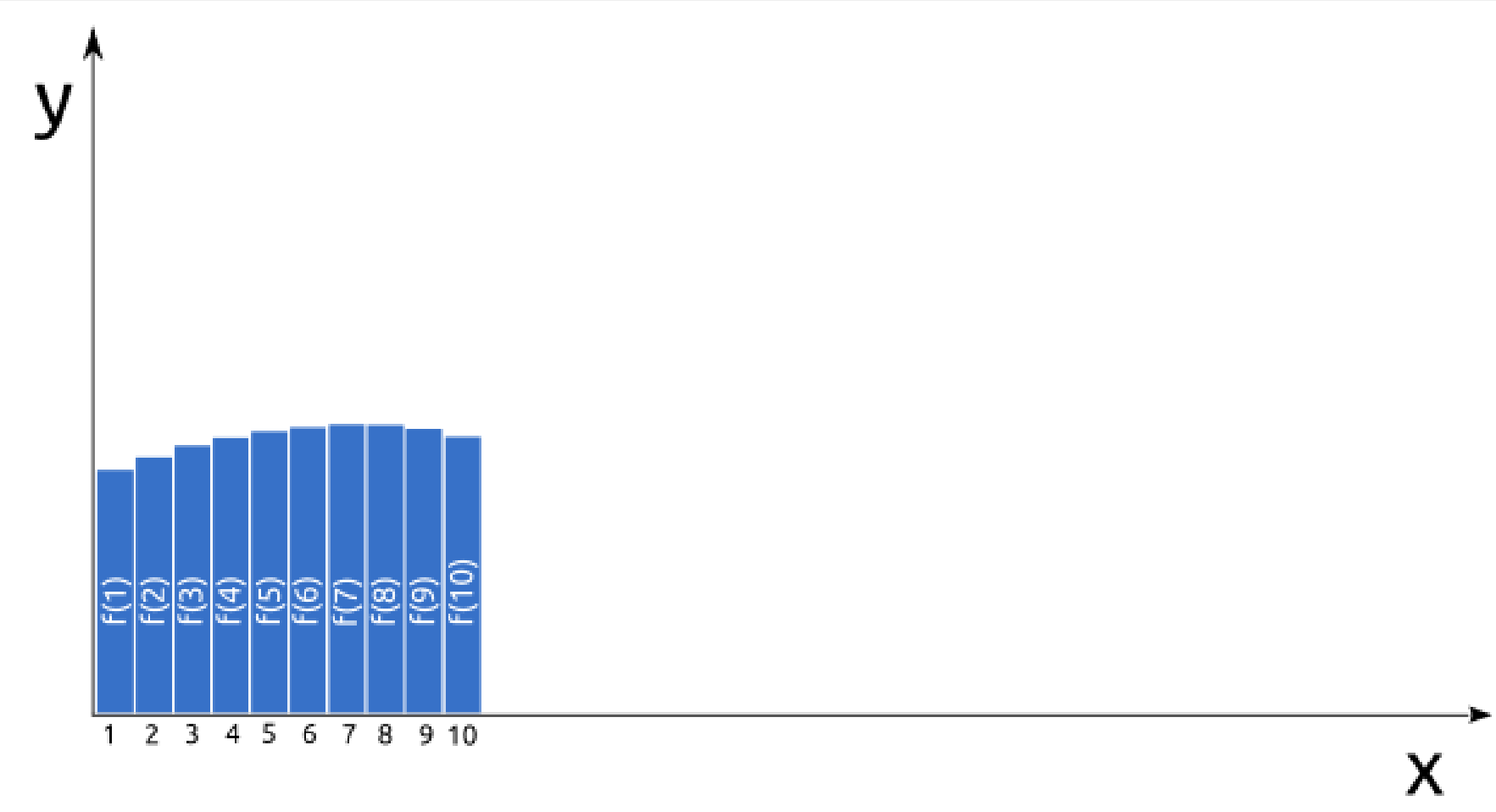

y
f(1)
f(2)
f(3)
f(4)
f(5)
1 2 3 4 5
x

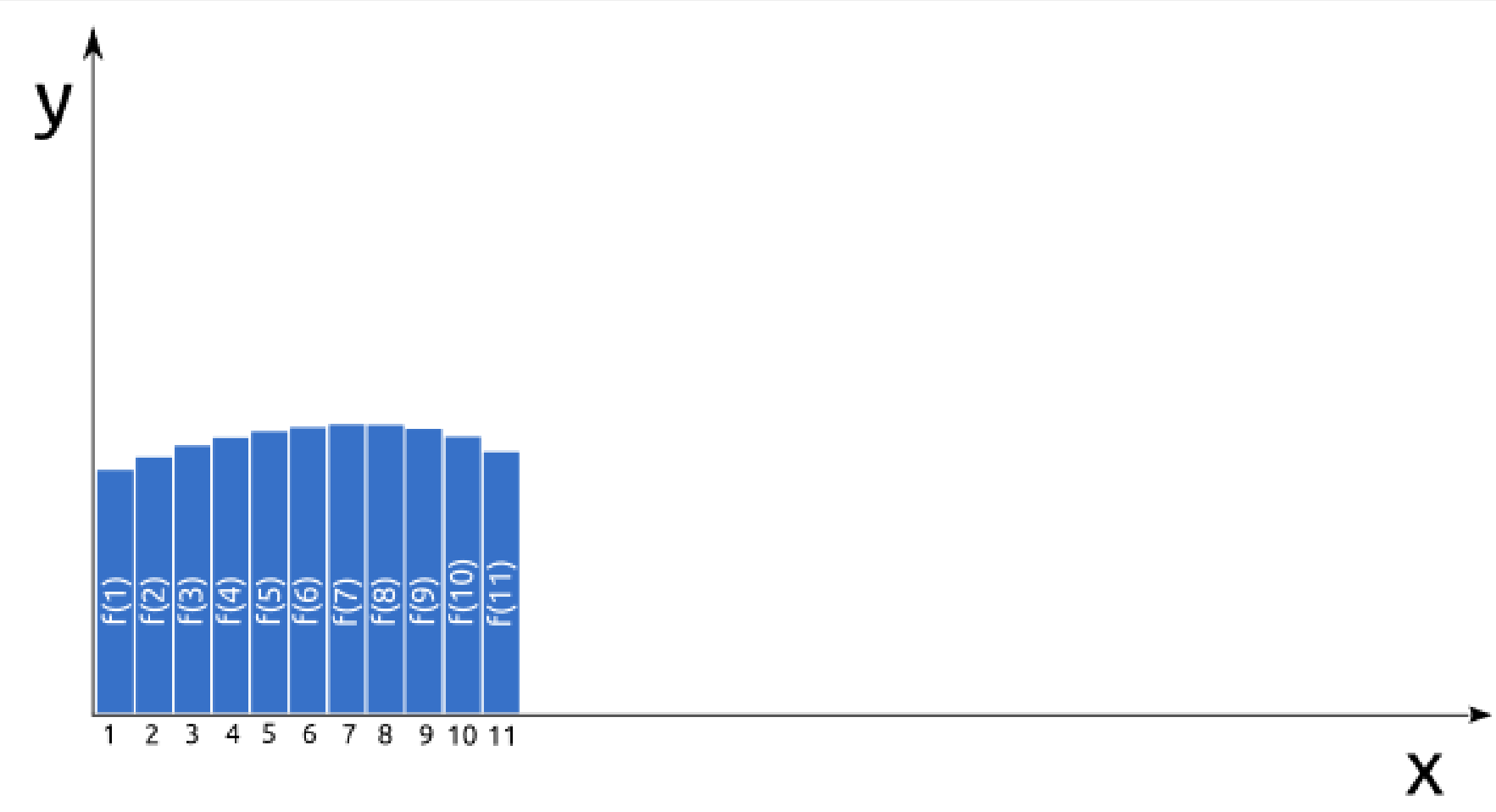

y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
1 2 3 4 5 6
x

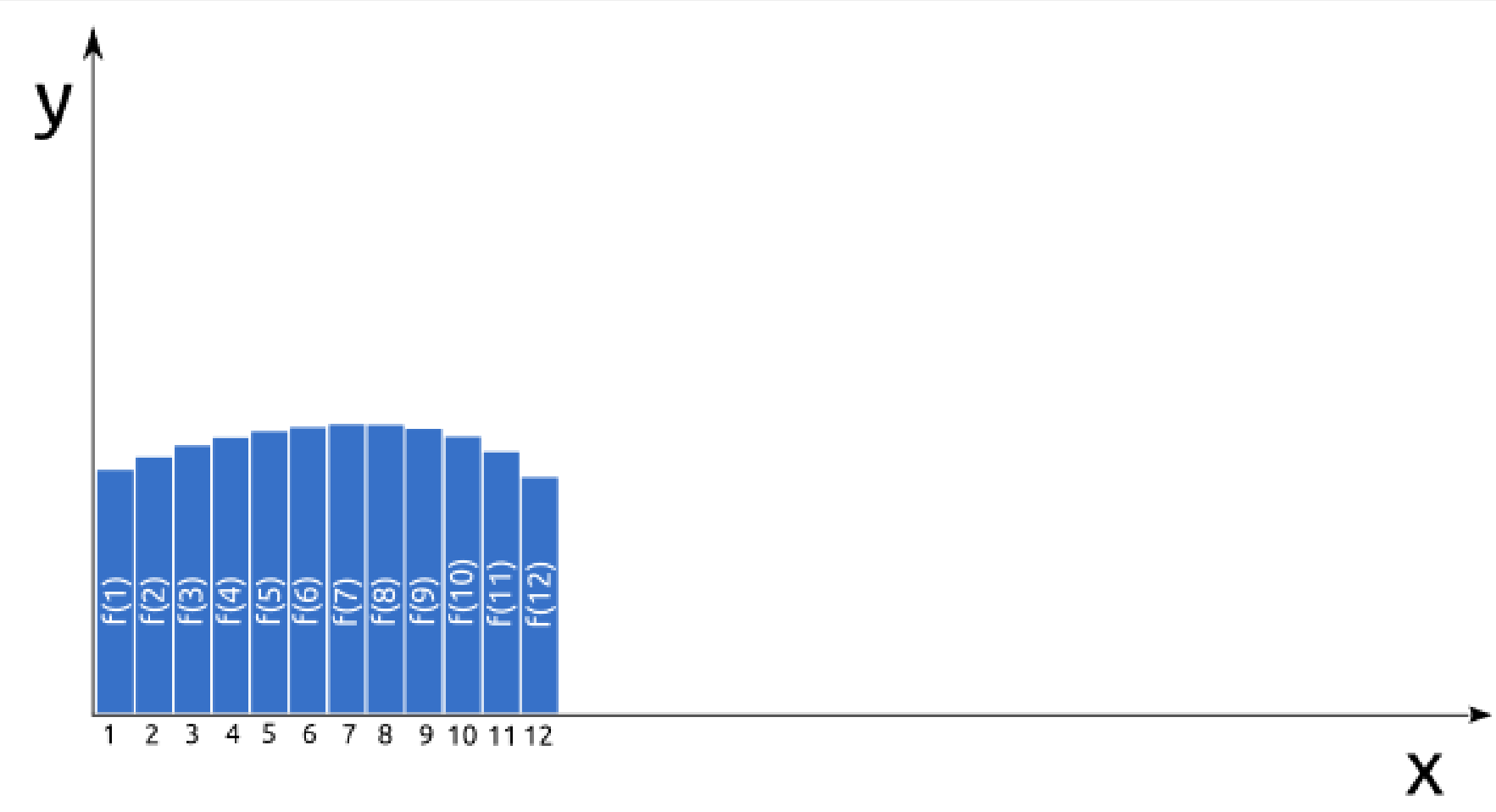

y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
1 2 3 4 5 6 7
x

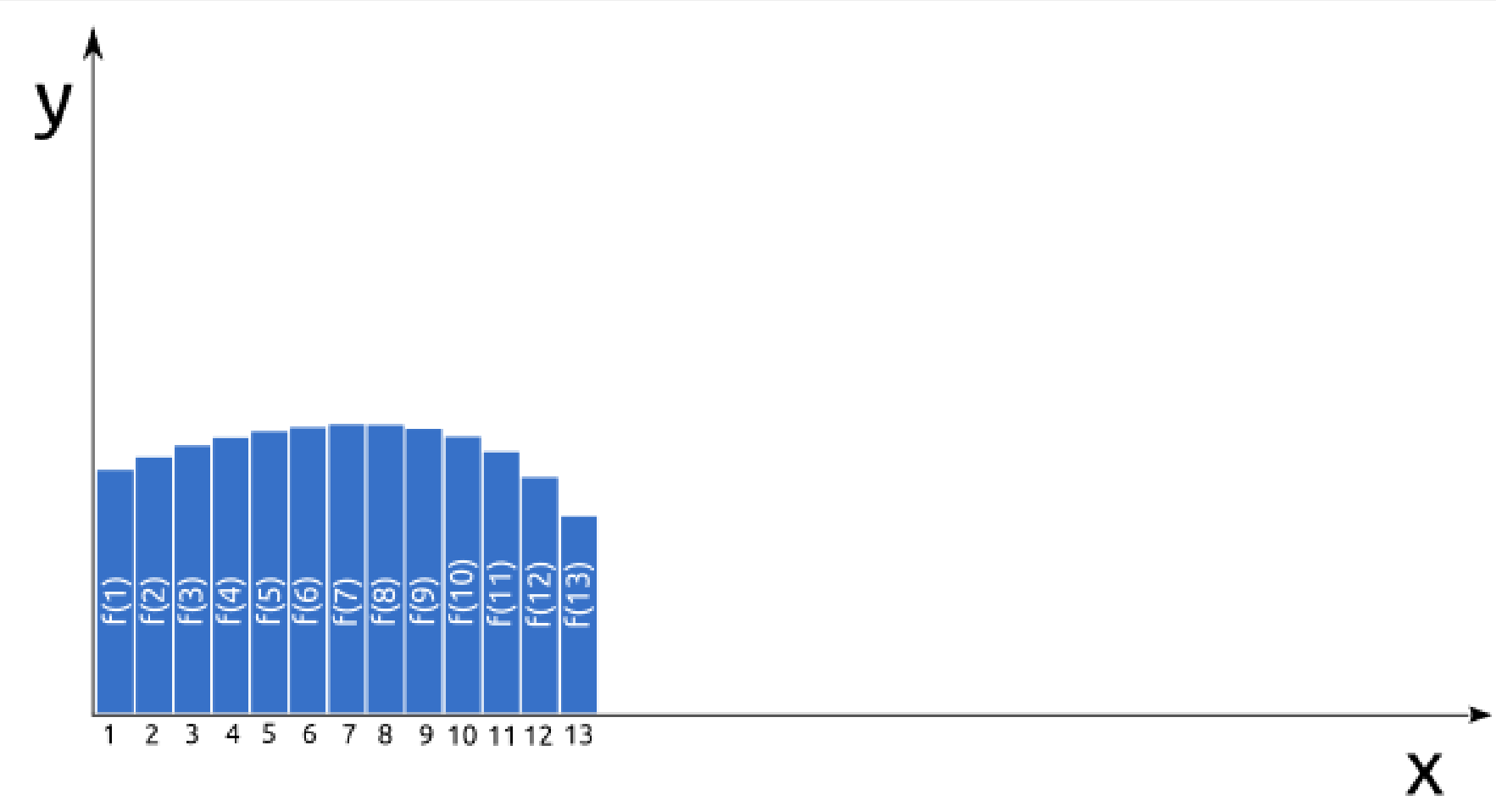

y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
1 2 3 4 5 6 7 8
x

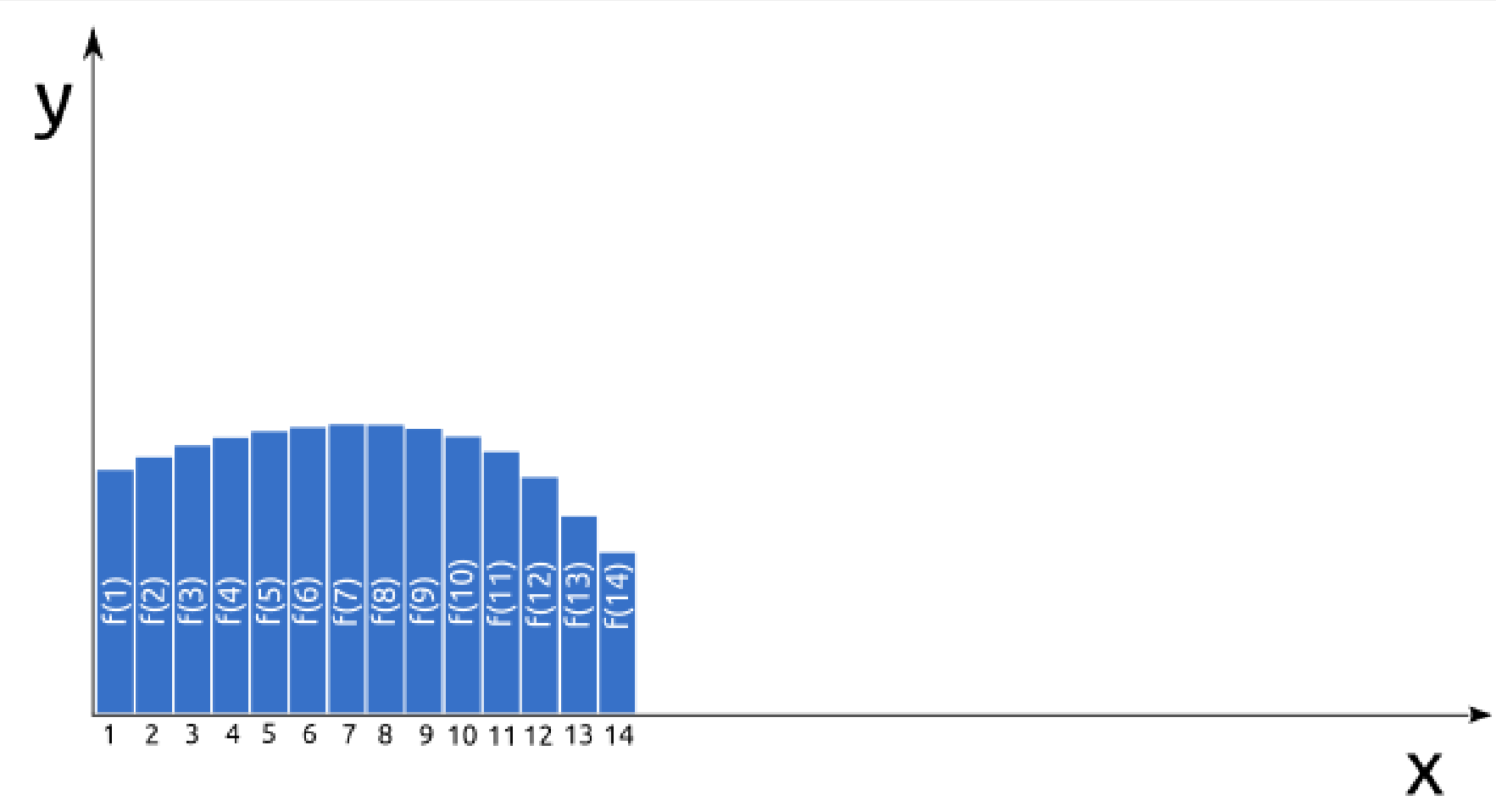

y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
1 2 3 4 5 6 7 8 9
x

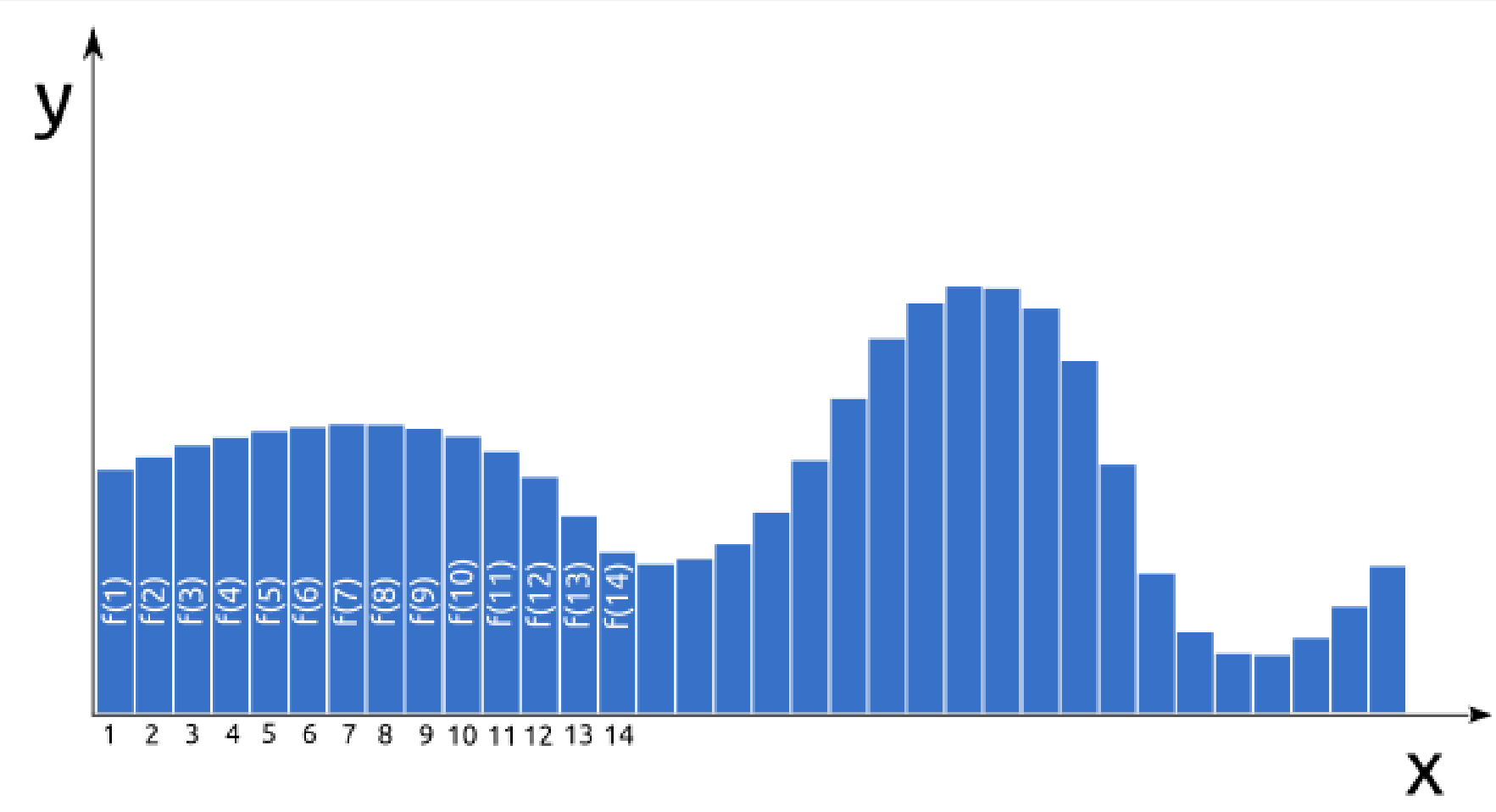

y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
1 2 3 4 5 6 7 8 9 10
x

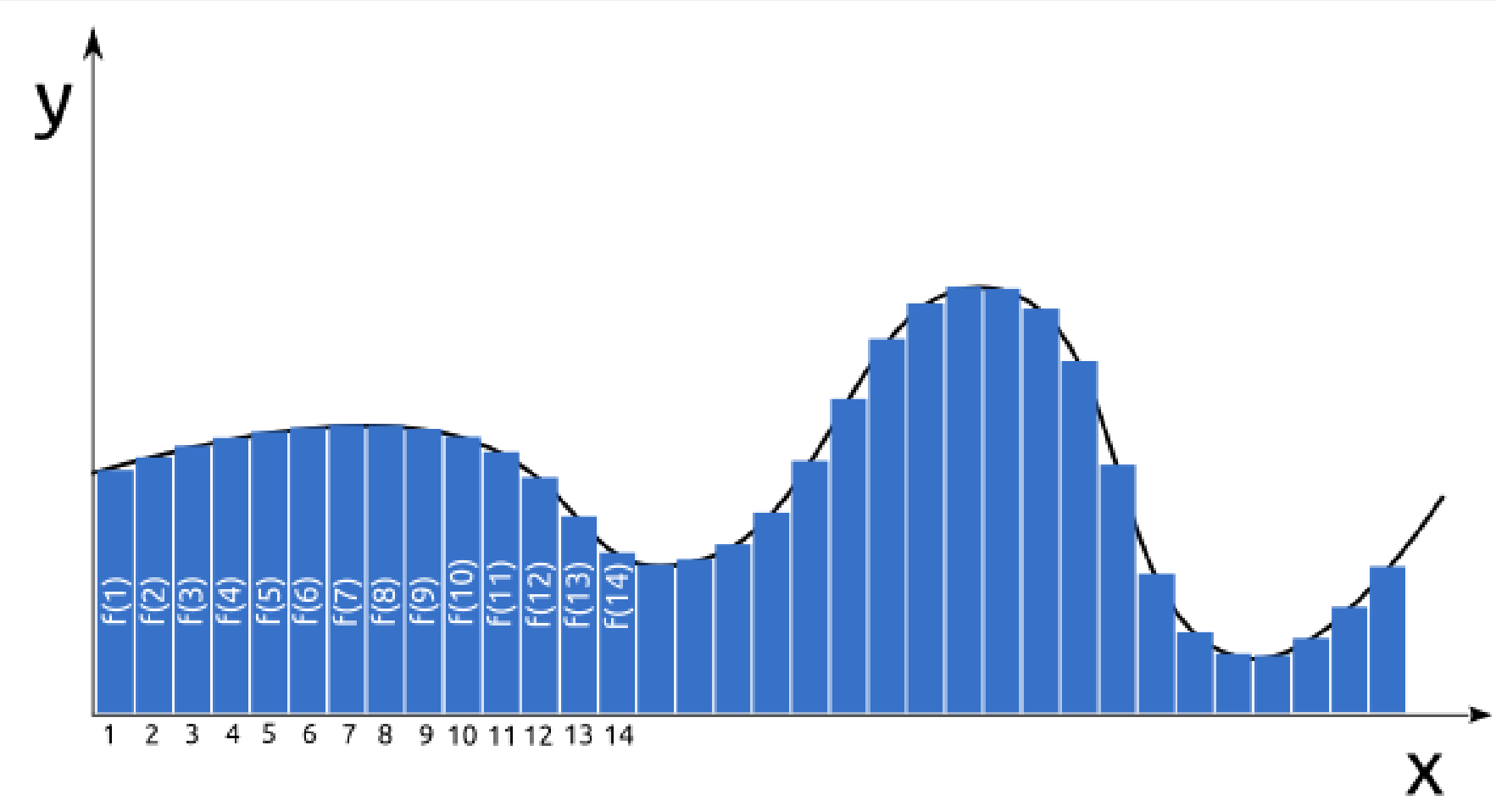

y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
f(11)
1 2 3 4 5 6 7 8 9 10 11
x


y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
f(11)
f(12)
1 2 3 4 5 6 7 8 9 10 11 12
x


y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
f(11)
f(12)
f(13)
1 2 3 4 5 6 7 8 9 10 11 12 13
x


y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
f(11)
f(12)
f(13)
f(14)
1 2 3 4 5 6 7 8 9 10 11 12 13 14
x


y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
f(11)
f(12)
f(13)
f(14)
1 2 3 4 5 6 7 8 9 10 11 12 13 14
x


y
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
f(8)
f(9)
f(10)
f(11)
f(12)
f(13)
f(14)
1 2 3 4 5 6 7 8 9 10 11 12 13 14
x

Траектория

Траекторията на тяло, изстреляно под ъгъл α спрямо земната повърхност с начална скорост V се описва от следните две уравнения:

$$x = V.cos(\alpha).t$$

$$y = V.sin(\alpha).t - \frac{1}{2}g.t^2$$

Траектория

Реализирайте клас Projectile, моделиращ снаряд, изстрелян във въздуха. Снарядът има маса, x и y координати. Реализирайте следните методи: конструктор с маса на снаряда и x координата (y е нула) метод move, който придвижва снаряда на време t. Той променя x и y. Метод shoot с параметри ъгъл и начална скорост. Методът да вика move през 0.1 секунди, докато y-координатата не стане 0. Нека програмата ви прави инстанция на такъв клас, като пита потребителя за ъгъл и начална скорост и след това извиква shoot. Текущите координати да се разпечатват във вид, удобен за внасяне в електронна таблица и изчертаването им.

Thanks for using Focus!

References